UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Diseño de una solución IoT para la gestión de reservas de puntos de carga para vehículos eléctricos

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Marian Cociu, Catalin

Tutor/a: Fons Cors, Joan Josep

CURSO ACADÉMICO: 2022/2023

Resumen

En los últimos años, la migración hacia la movilidad eléctrica y la adopción creciente de vehículos eléctricos han provocado un profundo cambio en la industria del transporte y la logística, junto con el deseado impacto medioambiental que esto conlleva. Sin embargo, la movilidad eléctrica todavía enfrenta desafíos importantes que limitan su plena expansión.

Uno de los desafíos clave de la movilidad eléctrica radica en la autonomía reducida de los vehículos eléctricos en comparación con sus contrapartes de combustión interna. Además, la velocidad más lenta de carga de las baterías y la infraestructura de carga en desarrollo dificultan la gestión y optimización colectiva de las operaciones de carga de vehículos, lo que ralentiza su adopción generalizada.

Este proyecto tiene como objetivo abordar esta problemática mediante la aplicación de estrategias tecnológicas modernas y principios de la Internet de las Cosas (IoT). Se propone diseñar un sistema que permita la comunicación máquina a máquina (M2M) a través de IoT entre vehículos y estaciones de carga, introduciendo el concepto de "reserva de carga".

El escenario se ambienta en una empresa de reparto que emplea vehículos eléctricos para atender las solicitudes de entrega de clientes. El objetivo es minimizar el tiempo de inactividad de los vehículos, que no puedan realizar entregas debido a baterías descargadas, coordinando sus actividades con las estaciones de carga. Para lograr esto, se introduce el concepto de "reserva de carga". Por un lado, las estaciones de carga ofrecen la opción de "reservarse" para un vehículo en particular durante un período definido, excluyendo otros vehículos. Por otro lado, los vehículos, cuando sus niveles de batería caen por debajo de un umbral, solicitan una carga a través de un servicio de gestión. Este servicio busca una estación de carga cercana y disponible, creando una reserva temporal para el vehículo. Una vez que este llega a la estación de carga, se autentica utilizando la plataforma IoT y, si se valida la reserva, el proceso de carga comienza.

El diseño de la solución se basará en una serie de enfoques y herramientas:

- Se aplicará la estrategia de Arquitectura Hexagonal para obtener una estructura flexible y desacoplada que facilite el despliegue como microservicios y su escalabilidad.
- La simulación del comportamiento de vehículos, cargadores y clientes se logrará mediante microservicios implementados en Spring Boot y Docker, orquestados por contenedores (Docker Swarm o Kubernetes). Esto permitirá un entorno controlado y escalable, con simulaciones realistas a través del ajuste de la cantidad de componentes en cualquier momento.
- El sistema empleará comunicaciones IoT a través de interfaces REST y MQTT para la interacción entre microservicios y dispositivos IoT. Se evaluará la viabilidad de plataformas IoT como TheThingsBoard o AWS IoT para desplegar dispositivos y facilitar comunicaciones, monitoreo y análisis de datos generados por los dispositivos a lo largo del tiempo.

Palabras clave: Movilidad Eléctrica, Electrolineras, Internet de las Cosas, Microservicios, Arquitectura Hexagonal

Resum

En els últims anys, la migració cap a la mobilitat elèctrica i l'adopció creixent de vehicles elèctrics han provocat un profund canvi en la indústria del transport i la logística, juntament amb l'impacte mediambiental desitjat que això comporta. No obstant això, la mobilitat elèctrica encara s'enfronta a reptes importants que limiten la seva plena expansió.

Un dels reptes clau de la mobilitat elèctrica rau en l'autonomia reduïda dels vehicles elèctrics en comparació amb els seus homòlegs de combustió interna. A més, la velocitat més lenta de càrrega de les bateries i la infraestructura de càrrega en desenvolupament dificulten la gestió i optimització col·lectiva de les operacions de càrrega de vehicles, la qual cosa retarda la seva adopció generalitzada.

Aquest projecte té com a objectiu abordar aquesta problemàtica mitjançant l'aplicació d'estratègies tecnològiques modernes i principis de la Internet de les Coses (IoT). Es proposa dissenyar un sistema que permeti la comunicació màquina a màquina (M2M) a través de l'IoT entre vehicles i estacions de càrrega, introduint el concepte de "reserva de càrrega".

L'escenari es desenvolupa en una empresa de repartiment que fa servir vehicles elèctrics per atendre les sol·licituds d'entrega de clients. L'objectiu és minimitzar el temps d'inactivitat dels vehicles, de manera que no puguin fer entregues a causa de les bateries descarregades, coordinant les seves activitats amb les estacions de càrrega. Per aconseguir això, es presenta el concepte de "reserva de càrrega". D'una banda, les estacions de càrrega ofereixen l'opció de "reservarse"per a un vehicle en particular durant un període definit, excloent altres vehicles. D'altra banda, quan els nivells de bateria dels vehicles cauen per sota d'un llindar, aquests sol·liciten una càrrega mitjançant un servei de gestió. Aquest servei busca una estació de càrrega propera i disponible, creant una reserva temporal per al vehicle. Una vegada que el vehicle arriba a l'estació de càrrega, s'autentica mitjançant l'IoT i, si es valida la reserva, comença el procés de càrrega.

El disseny de la solució es basarà en una sèrie d'enfocaments i eines:

- S'aplicarà l'estratègia de l'Arquitectura Hexagonal per obtenir una estructura flexible i desacoblat que faciliti el desplegament com a microserveis i la seva escalabilitat.

- La simulació del comportament de vehicles, càrregues i clients s'aconseguirà mitjançant microserveis implementats en Spring Boot i Docker, orquestrats per contenidors (Docker Swarm o Kubernetes). Això permetrà un entorn controlat i escalable, amb simulacions realistes mitjançant l'ajust de la quantitat de components en qualsevol moment.

- El sistema farà servir comunicacions IoT a través d'interfícies REST i MQTT per a la interacció entre microserveis i dispositius IoT. Es valorarà la viabilitat de plataformes IoT com TheThingsBoard o AWS IoT per desplegar dispositius i facilitar comunicacions, monitoratge i anàlisi de dades generades pels dispositius al llarg del temps.

Paraules clau: Mobilitat Elèctrica, Estacions de Càrrega per a Vehicles Elèctrics (també conegudes com a Estacions de Càrrega EV), Internet de les Coses (IoT), Microserveis, Arquitectura Hexagonal

Abstract

In recent years, the shift towards electric mobility and the growing adoption of electric vehicles have led to a profound change in the transportation and logistics industry, along with the desired environmental impact that comes with it. However, electric mobility still faces significant challenges that limit its full expansion.

One of the key challenges of electric mobility lies in the reduced range of electric vehicles compared to their internal combustion counterparts. Additionally, the slower battery charging speed and the ongoing development of charging infrastructure make collective management and optimization of vehicle charging operations more complex, thus slowing down widespread adoption.

This project aims to address this issue through the application of modern technological strategies and principles of the Internet of Things (IoT). The objective is to design a system that enables machine-to-machine (M2M) communication through IoT between vehicles and charging stations, introducing the concept of "charging reservation."

The scenario is set in a delivery company that uses electric vehicles to fulfill customer delivery requests. The goal is to minimize vehicle downtime caused by depleted batteries by coordinating their activities with charging stations. To achieve this, the concept of "charging reservation" is introduced. On one hand, charging stations offer the option to "reserve" themselves for a specific vehicle during a defined period, excluding other vehicles. On the other hand, when vehicle battery levels drop below a threshold, they request charging through a management service. This service searches for a nearby and available charging station, creating a temporary reservation for the vehicle. Once the vehicle arrives at the charging station, it authenticates through IoT, and if the reservation is validated, the charging process begins.

The solution's design will be based on a series of approaches and tools:

- The Hexagonal Architecture strategy will be applied to achieve a flexible and decoupled structure that facilitates despliegue as microservices and scalability.
- Simulating the behavior of vehicles, chargers, and clients will be achieved through microservices implemented in Spring Boot and Docker, orchestrated by containers (Docker Swarm or Kubernetes). This allows for a controlled and scalable environment, with realistic simulations by adjusting the number of components at any given time.

- The system will employ IoT communications through REST and MQTT interfaces for interaction between microservices and IoT devices. The feasibility of IoT platforms like TheThingsBoard or AWS IoT will be evaluated to deploy devices and facilitate communications, monitoring, and analysis of data generated by devices over time.

Key words: Electric Mobility, Electric Vehicle Charging Stations (also known as EV Charging Stations), Internet of Things (IoT), Microservices, Hexagonal Architecture

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

Uno de los mayores problemas presentes en la actualidad es la contaminación, se han propuesto y llevado a cabo diferentes soluciones, entre ellas, la electrificación de los medios de transporte, principalmente en vehículos terrestres como coches o furgonetas. Esto ha generado un cambio significativo en la manera de desplazarse para aquellos que hayan adoptado este tipo de vehículos. Ese cambio es la necesidad de una realizar una planificación más rigurosa de los viajes, sobretodo en aquellos de largas distancias dada la autonomía más reducida de este tipo vehículos si son comparados con sus homólogos que utilizan un motor de combustión interna. La tabla siguiente muestra algunos de los coches con mejor autonomía en el momento actual:

MODELO DE COCHE AUTONOMÍA	(KM)
Lucid Air Grand Touring	830
Mercedes EQS 450+ AMG Line	770
Tesla Model S	663
Mercedes-Benz EQE 300 AMG	660
Fisker Ocean 'Extreme'	640
BMW iX xDrive50	630
BMW i7 xDrive60	600
Tesla Model 3	568

Tabla 1.1: Autonomía de algunos vehículos eléctricos de 2023

Otro problema muy importante además de la autonomía es la escasez de puntos de carga, ya que la red de electrolineras todavía está en una etapa de expansión, reforzando la necesidad mencionada anteriormente.

Los problemas mencionados previamente hacen que las empresas de las industrias del transporte y la logística se enfrenten a desafíos cada vez mayores para adaptarse a esta nueva realidad y optimizar sus operaciones y al mismo tiempo ofrecer servicios eficientes y sostenibles. Para cumplir con sus objetivos, estas empresas deben planificar minuciosamente todas sus rutas de reparto e integrar la necesidad de recargar sus vehículos de manera periódica. Por tanto se propone como solución integrar la asignación de repartos entrantes por parte de los clientes dentro de un sistema que gestione las reservas de puntos de carga de manera automática.

1.1 Motivación

La elección de este tema viene motivada en gran parte por la realización de un curso que trata las herramientas de *software* más esenciales en la transición energética. Entre los contenidos impartidos había dos módulos que me resultaron llamativos, uno de ellos hablando de la comunicación entre un sistema central y puntos de carga y el otro sobre microservicios. Este tema brinda la oportunidad no solo de profundizar y poner en práctica gran parte de los conocimientos adquiridos durante el grado y el curso, sino también de aprender y utilizar algunas de las tecnologías más demandadas en la actualidad. Este trabajo ofrece una experiencia valiosa tanto para un estudiante, así como para un futuro profesional dado que fortalecerá las habilidades técnicas ya adquiridas y ganando experiencia práctica al mismo tiempo.

1.2 Objetivos

Como bien se ha mencionado al final de la primera parte de este capítulo, los principales objetivos de la solución a desarrollar serán:

- Gestionar las reservas de puntos de carga de manera automática.
- Optimizar la asignación de repartos a los vehículos disponibles para maximizar el tiempo operativo de la flota.

Observando los objetivos anteriores, podemos apreciar que tanto los vehículos, clientes, así como los cargadores son piezas fundamentales del sistema, por tanto definiremos algunos objetivos adicionales:

- Desarrollar unos microservicios que simulen el comportamiento de los componentes mencionados anteriormente.
- Establecer las diferentes comunicaciones entre los microservicios.
- Integrar una plataforma IoT para la monitorización y análisis de los datos que vehículos y puntos de carga van generando en tiempo real.
- Poner en producción el sistema utilizando un orquestador de contendores de forma local e idealmente en un proveedor de servicios en la nube.

1.3 Metodología

Como metodología se ha optado por la metodología clásica o en cascada ya que se considera más adecuada dada la complejidad del sistema. Se ha descartado cualquier tipo de metodología ágil ya que el software a desarrollar requiere de una fase de diseño e implementación sólidas, las cuales son incompatibles con el desarrollo incremental y evolutivo. Ahora bien, la aplicación del modelo en

cascada no será en su forma estricta dado que se considera más adecuado probar las funcionalidades de manera gradual durante el desarrollo.

Desde la fecha de alta de este trabajo hasta la fecha máxima de entrega se dispone de 11 semanas para finalizar el trabajo, estas semanas se han repartido de la siguiente forma:

1. Análisis: 1 semanas
2. Diseño: 2 semanas
3. Implementación y Pruebas: 7 semanas
4. Implantación: 1 semana

1.4 Estructura de la memoria

La presente memoria consta de ocho capítulos:

- **Introducción**. Se expone la motivación del trabajo y los objetivos que se han propuesto.
- **Contexto tecnológico**. Trata temas recurrentes en el desarrollo de soluciones de la temática del trabajo, soluciones existentes y donde encaja la solución a desarrollar.
- **Análisis del problema**. Se definen las funcionalidades de los componentes a desarrollar así como otros factores influyentes en la posterior fase de diseño.
- **Diseño de la solución**. Se presenta la arquitectura global del sistema y la interna de algunos componente. Adicionalmente se detalla el diseño de las comunicaciones.
- **Tecnología utilizada**. Este capítulo expone todas las tecnologías seleccionadas para la implementación de la solución.
- **Desarrollo de la solución propuesta**. Se muestra la aplicación del patrón arquitectónico seleccionado detalladamente así como otros fases del desarrollo.
- **Implantación**. Se muestran las maneras de desplegar la solución así como la solución a un problema recurrente en esta fase.
- **Conclusiones**. Se comparan los objetivos alcanzados con los enumerados al principio de este trabajo, se determina la relación del trabajo con los estudios y se abren otras líneas de desarrollo futuras.

CAPÍTULO 2
Contexto tecnológico

2.1 Microservicios

Los microservicios son una arquitectura de *software* en la que una aplicación se divide en múltiples servicios más pequeños e independientes. Cada uno de estos servicios se encarga de una funcionalidad específica y puede ser desarrollado y desplegado de forma independiente. Esto contrasta con la arquitectura monolítica, en la que una aplicación se construye como una única unidad unificada.

Hay varias ventajas en el uso de microservicios en comparación con la arquitectura monolítica. Una de las principales ventajas es la escalabilidad. Cada microservicio puede escalarse de forma independiente, lo que permite a los desarrolladores ajustar el rendimiento de cada servicio según sea necesario. Esto también permite una mayor flexibilidad en el despliegue, ya que los servicios pueden desplegarse en diferentes entornos y plataformas.

Otra ventaja es la facilidad de desarrollo y mantenimiento. Al dividir una aplicación en múltiples servicios más pequeños, los desarrolladores pueden trabajar en cada servicio de forma independiente, lo que reduce la complejidad y facilita el desarrollo y mantenimiento. Además, los microservicios permiten una mayor modularidad, lo que facilita la reutilización de código y reduce el acoplamiento entre componentes.

Sin embargo, también hay desventajas en el uso de microservicios. Una de las principales desventajas es la complejidad adicional que se introduce al tener múltiples servicios en lugar de una única aplicación monolítica. Esto puede aumentar el tiempo necesario para desarrollar y desplegar la aplicación, así como aumentar la complejidad en la gestión y monitorización de los servicios.

Otra desventaja es el aumento del riesgo de fallos en cascada. Si un servicio falla, puede afectar a otros servicios que dependen de él, lo que puede provocar fallos en cascada en toda la aplicación. Por lo tanto, es importante diseñar los microservicios con tolerancia a fallos y redundancia para minimizar este riesgo.

Además de las ventajas y desventajas mencionadas anteriormente, hay otros factores a considerar al elegir entre microservicios y arquitectura monolítica. Uno de estos factores es el tamaño y complejidad de la aplicación. Las aplicaciones grandes y complejas pueden beneficiarse del uso de microservicios, ya que esto permite dividir la aplicación en componentes más pequeños y manejables. Sin

embargo, para aplicaciones más pequeñas y sencillas, puede ser más eficiente utilizar una arquitectura monolítica.

Otro factor a considerar es el equipo de desarrollo. Los equipos grandes y distribuidos pueden beneficiarse del uso de microservicios, ya que esto permite a cada equipo trabajar en un servicio específico sin interferir con el trabajo de otros equipos. Sin embargo, para equipos más pequeños o centralizados, puede ser más eficiente utilizar una arquitectura monolítica.

En conclusión, los microservicios son una arquitectura de *software* poderosa que ofrece varias ventajas sobre la arquitectura monolítica. Sin embargo, también presentan desafíos adicionales y no son adecuados para todas las aplicaciones o equipos de desarrollo. La elección entre microservicios y arquitectura monolítica dependerá de varios factores, como el tamaño y complejidad de la aplicación y el equipo de desarrollo.

2.2 Internet de las cosas

Este concepto fue utilizado por primera vez a finales de los años 90 por Kevin Ashton[4], con la finalidad de describir un sistema donde el mundo físico estaba conectado a Internet a través de sensores, incluyendo las etiquetas RFID(*Radio Frequency Identification*) para la gestión de mercancías en las cadenas de suministro. El uso de este tipo de sistemas, sin embargo, es anterior a la descripción utilizada por Ashton, remontándose a la década de 1970, cuando *Theodore Paraskevakos* llevó a la práctica un prototipo para el reconocimiento de llamadas telefónicas y posteriormente en 1977 fundó la empresa*Metretek, Inc* para realizar lecturas de medidores automáticos comerciales a través de una red telefónica y gestión de carga para servicios eléctricos. Estos dos hitos fueron los que acabaron de asentar las bases de la comunicación máquina a máquina(M2M)[5], que avanzó a pasos agigantados gracias a los avances en la telefonía y la conexión inalámbrica a Internet. La principal diferencia con el internet de las cosas radica en la forma de conseguir la conectividad, utilizando este último protocolos estándar basadas en redes IP.

El Internet de las cosas es una red que conecta millones de dispositivos como sensores, vehículos e incluso edificios, a Internet con distintas finalidades, ya sea proveer información o realizar una determinada acción. La idea de conectar diferentes tipos de dispositivos a Internet ofrece la posibilidad de crear aplicaciones que faciliten ciertas tareas y mejoren la calidad de vida de las personas en diversas áreas. Todas estas aplicaciones se pueden agrupar en una serie de casos de usos comunes que se detallarán en la sección siguiente.

2.3 Casos de uso de IoT

Los casos de uso más populares son los siguientes:

- **Automatización del hogar**: los dispositivos IoT se pueden utilizar para automatizar tareas en el hogar, como controlar las luces, el termostato y los electrodomésticos.
- **Atención sanitaria**: los dispositivos IoT se pueden utilizar para recopilar datos sobre la salud de las personas, como la frecuencia cardíaca, la presión arterial y la actividad física.
- **Mantenimiento predictivo**: los dispositivos IoT se pueden utilizar para monitorizar el rendimiento de los equipos y predecir posibles fallos.
- **Logística**: los dispositivos IoT se pueden utilizar para rastrear el movimiento de productos y mercancías.
- **Seguridad**: los dispositivos IoT se pueden utilizar para mejorar la seguridad de las personas y los bienes.
- **Agricultura:** los dispositivos IoT se pueden utilizar para monitorizar el clima, las condiciones del suelo y el crecimiento de las plantas. Esto puede ayudar a los agricultores a mejorar la productividad y la eficiencia.
- **Industria**: los dispositivos IoT se pueden utilizar para controlar los procesos de fabricación y garantizar la calidad de los productos. Esto puede ayudar a las empresas a reducir costes y mejorar la seguridad.
- **Transporte**: los dispositivos IoT se pueden utilizar para mejorar la seguridad del tráfico y optimizar la eficiencia del transporte. Esto puede ayudar a reducir las emisiones de gases de efecto invernadero y mejorar la calidad del aire.
- **Ciudad inteligente**: los dispositivos IoT se pueden utilizar para mejorar la calidad de vida de los ciudadanos. Esto puede incluir aplicaciones para la gestión del tráfico, el alumbrado público, la recogida de residuos y la seguridad ciudadana.

2.4 Plataformas IoT

Las plataformas IoT más completas son aquellas que ofrecen un conjunto completo de funciones para el desarrollo, implementación y gestión de soluciones IoT. Estas plataformas suelen proporcionar las siguientes funcionalidades:

- **Gestión de dispositivos**: La capacidad de conectar, gestionar y supervisar un gran número de dispositivos IoT.
- **Almacenamiento y análisis de datos**: La capacidad de almacenar y analizar los datos recopilados por los dispositivos IoT.
- **Desarrollo de aplicaciones**: La capacidad de desarrollar aplicaciones IoT personalizadas.
- **Integración con otros sistemas**: La capacidad de integrar las soluciones IoT con otros sistemas, como sistemas empresariales o sistemas de TI.

2.4.1. Plataformas IoT privativas

- Microsoft Azure IoT Suite[6]: Esta plataforma ofrece una amplia gama de funcionalidades, incluyendo redes privadas, gestión de dispositivos, análisis de datos e integración con aplicaciones empresariales.
- Amazon Web Services (AWS) IoT[7]: AWS IoT ofrece una gran variedad de servicios para la implementación de soluciones IoT, incluyendo redes privadas, gestión de dispositivos, análisis de datos e integración con aplicaciones empresariales.
- IBM Watson Iot Platform[8]: Proporciona un conjunto de herramientas y servicios diseñados para ayudar a las empresas a conectar, gestionar, analizar y aprovechar los datos generados por dispositivos IoT de manera efectiva.

2.4.2. Plataformas IoT de código abierto

- ThingsBoard[9]: Esta plataforma ofrece una amplia gama de funcionalidades, incluyendo gestión de dispositivos, análisis de datos, visualización de datos e integración con aplicaciones empresariales.
- OpenRemote[10]: Esta plataforma ofrece una solución completa para el desarrollo de soluciones IoT, incluyendo gestión de dispositivos, análisis de datos, visualización de datos e integración con aplicaciones empresariales.
- Kaa[11]: Es una plataforma de código abierto que proporciona una serie de servicios para el desarrollo de soluciones IoT.

2.5 Soluciones existentes

El sistema a desarrollar compite en tres frentes con otras soluciones que ya llevan un tiempo en el mercado, aunque algunas de ellas son relativamente recientes. Las áreas en las que se compite son las siguientes:

- Reservas de puntos carga: Este concepto es relativamente nuevo, a pesar de eso ya existen algunas soluciones en este ámbito, algunas de las más populares son *Electromaps[12], SmartMobility[13].* Principalmente constan de un mapa que muestra los diferentes puntos de carga y su disponibilidad, la funcionalidad de filtrar y reservar los cargadores así como la posibilidad de pagar directamente desde la propia aplicación.
- Monitorización de datos de vehículos: este tipo de aplicaciones son conocidas como gestores de flotas de vehículos eléctricos. Las principales características son la visualización de la posición y porcentaje de batería de los vehiculos, posición y disponibilidad de los cargadores(no pudiendo reservarlos), estadísticas y planificación de cargas entre otras. Algunos ejemplos de estos gestores son *EVectrum[14], EVcharge[15], chargepoint[16].*

- Planificar y asignar repartos: Generalmente hay pocas aplicaciones que incluyan ambas cosas, ya que la mayoría se centra solamente en la optimización de las rutas, la solución más completa encontrada ha sido *DispatchTrack[17].*

2.6 Conclusiones

Este proyecto no pretende competir en contra de ninguna de las herramientas mencionadas anteriormente de manera específica, sino que pretende integrar las principales funcionalidades de cada una en un solo sistema. Para ser más específicos, el sistema a desarrollar debe ser capaz de reservar un punto de carga de manera automática según el porcentaje de batería de cada vehículo, monitorizar el estado en tiempo real de cada uno de esos componentes y de poder gestionar todos los repartos actuales y los entrantes.

CAPÍTULO 3
Análisis del problema

Tal y como se ha mencionado en el capítulo anterior, el proyecto a desarrollar tiene tres grandes objetivos, estos objetivos, así como las funcionalidades que presentan algunas de las herramientas descritas anteriormente y la necesidad de simular el comportamiento de vehículos y cargadores, serán la base para la especificación de requisitos que sigue a continuación.

3.1 Definición de requisitos

Para especificar los requisitos primero se ha optado por estudiar el dominio del problema y los principales competidores para tener una mejor comprensión de lo que un posible usuario se esperaría de nuestra solución. Para la definición de requisitos se ha elegido la técnica de lluvia de ideas o *brainstorming*, la especificación se realizará en un grado informal, es decir, en lenguaje natural. A grandes rasgos, con el sistema van a interactuar tres grandes actores:

- Punto/s de carga
- Clientes de reparto
- Vehículos

Estos actores al ser componentes a desarrollar necesitarán de su propia especificación de requisitos, las cuáles se detallarán a continuación.

3.1.1. Punto de carga

Para simplificar la futura implementación, asumiremos que un punto de carga dispone solamente de un conector, así que no puede suministrar más un vehículo cada vez. Por tanto los requisitos finales serían:

1. Ser capaz de registrarse en el sistema de gestión.
2. Ser capaz de cargar un vehículo eléctrico.
3. Tener la capacidad de recibir una sola reserva cuando está disponible.

4. Tener la capacidad de rechazar una reserva entrante si ya se dispone de una.

5. Denegar la carga a cualquier vehículo que no sea el especificado en la reserva.

6. Completar la reserva actual cuando el vehículo haya terminado su carga y poder aceptar una nueva reserva nuevamente.

7. Enviar sus datos y estadísticas en tiempo real

3.1.2. Vehículo

En este caso, asumiremos que un vehículo solo dispone de espacio suficiente para un reparto cada vez, por tanto los requisitos que debe cumplir son:

1. Ser capaz de registrarse en el sistema de gestión.

2. Ser capaz de recibir un solo reparto cada vez que está disponible.

3. Desplazarse a la posición de recogida del reparto una vez recibido.

4. Desplazarse a la posición de entrega del reparto una vez recogido.

5. Notificar al sistema de gestión de que el reparto ha sido completado una vez ha llegado a su destino.

6. Solicitar una reserva de un punto de carga una vez llegue a cierto porcentaje de batería, por seguridad este será del 20 %.

7. Esperar en la misma posición hasta que no se le asigne una reserva, independientemente de si va a recoger o entregar un reparto.

8. Desplazarse a la posición del punto de carga una vez recibida la reserva.

9. Ser capaz de recibir la carga suministrada por el cargador una vez ha llegado a su posición.

10. Enviar sus datos y estadísticas en tiempo real, aun estando en reposo.

11. Enviar el estado del reparto al sistema de gestión de manera periódica.

3.1.3. Cliente de reparto

Por simplicidad no entraremos en detalles como podría ser número, dimensiones y peso del paquete. El único requisito de un cliente es:

- Solicitar un reparto con una posición de recogida y una posición de entrega, con la obligación de que ambas sean diferentes.

Si bien es cierto que algunos de los requisitos de las secciones anteriores son obvios, dada la naturaleza de estos actores, siguen siendo componentes a implementar. Por muy obvia que pueda resultar una cosa, sigue habiendo posibilidad de que esta se pueda olvidar. Siempre es conveniente tener precaución en esta fase del ciclo de vida del *software* ya que cualquier error descubierto en fases posteriores, como la de implementación o puesta en producción, tendrá asociado un coste mucho mayor.

3.1.4. Servicio de gestión de reservas y repartos

Los requisitos de este servicio son:

1. Registrar un punto de carga.
2. Registrar un vehículo.
3. Registrar los repartos entrantes de los clientes.
4. Completar un reparto, cuando un vehículo así lo indique.
5. Completar una reserva cuando un punto de carga así lo indique.
6. Asignar periódicamente los repartos no completados a los vehiculos disponibles.
7. Reservar un punto de carga en cuanto un vehículo lo solicite, siempre y cuando haya puntos de carga disponibles.
8. Actualizar el estado de los vehículos periódicamente.
9. Actualizar el estado de los puntos de carga periódicamente.
10. Actualizar el estado de los repartos periódicamente.

3.2 Identificación y análisis de las soluciones posibles

En esta sección veremos los factores más determinantes que influenciarán diseño final del sistema a implantar.

3.2.1. Tipos de comunicación

Analizando los requisitos especificados anteriormente podemos observar que el sistema a desarrollar tendrá principalmente dos tipos de comunicaciones:

1. **Telemetría**: se trata de una tecnología que permite la transmisión de datos de un dispositivo a otro de manera remota, típicamente en tiempo real. Un ejemplo adaptado a nuestro caso es el envío del estado de un vehículo al sistema de gestión o a una plataforma IoT. En este contexto específico, la

duplicidad o perdida de algunos mensajes es asumible. Dada la naturaleza de este tipo de comunicaciones, el protocolo a utilizar debe cumplir algunos requisitos imprescindibles:

- Compatibilidad con sistemas de bajos recursos, ya que gran parte de los mensajes enviados proviene de sensores o sistemas cuya eficiencia energética es crítica.
- Seguridad, factor decisivo si los datos son sensibles.
- Alto rendimiento en envío y recepción de mensajes.
- Escalabilidad, el sistema debe comportarse de la misma manera aunque el número de dispositivos incremente drásticamente.

Los protocolos de comunicación más populares utilizados en el IoT son MQTT(*Message Queuing Telemetry Transport*) y CoAP(*Constrained Application Protocol*).

La siguiente tabla resume las principales características de cada uno:

Tabla 3.1: Características de los protocolos de comunicación IoT

Protocolo	**Eficiencia energética**	**Seguridad**	**Rendimiento**	**Escalabilidad**
MQTT	Eficiente	Seguro	Rápido	Escalable
CoAP	Muy eficiente	Seguro	Medio	Escalable

MQTT utiliza un modelo de comunicación publicador/suscriptor, mientras que CoAP utiliza un modelo petición-respuesta. El modelo publicador/-suscriptor implica el uso de un *middleware* de mensajería, cosa que puede implicar cambios significativos en el diseño del sistema, discutiremos su importancia en la subsección siguiente.

2. **Llamadas a servicios**: Utilizadas cuando se requiera fiabilidad y la perdida de mensajes no es aceptable. Un ejemplo es la solicitud de reparto de un cliente o la confirmación de entrega de un reparto. Para este tipo de comunicaciones hay dos alternativas: Servicios de tipo REST(*Representational State Transfer*) o SOAP(*Simple Object Access Protocol*). Si bien es cierto que REST es un estilo arquitectónico web y SOAP es un protocolo de comunicación, ambas opciones pueden cumplir con el propósito requerido.

3.2.2. *Middleware*

El *middleware*[1] es un *software* que actúa como intermediario entre diferentes aplicaciones, sistemas operativos o componentes de *software* para facilitar la comunicación y la interacción entre ellos. Proporciona abstracciones y servicios que ayudan a simplificar la complejidad de la comunicación entre diferentes componentes de *software* y sistemas heterogéneos.Por todo lo anterior, desempeña un papel fundamental en la arquitectura de sistemas distribuidos y en la implementación de tecnologías como el Internet de las Cosas (IoT), la computación en la nube y la integración de sistemas empresariales.

Al usar un modelo publicador/suscriptor se simplifican en gran medida las comunicaciones, ya que no hay necesidad de que los distintos participantes se conozcan entre sí para recibir datos de otros. Los participantes que publiquen mensajes, lo harán sin conocer quien los va a recibir, mientras que los suscriptores de ciertos tópicos o colas recibirán los mensajes sin conocer directamente quien los ha publicado. Esto mejora la escalabilidad del sistema de forma importante y a la vez la eficiencia, ya que no se sobrecargan otros componentes para la redistribución de esos mensajes. Un *middleware* de mensajería también puede ofrecer medidas de seguridad para proteger la falsificación, manipulación y acceso no autorizado a los mensajes. La figura siguiente describe la situación mencionada anteriormente.

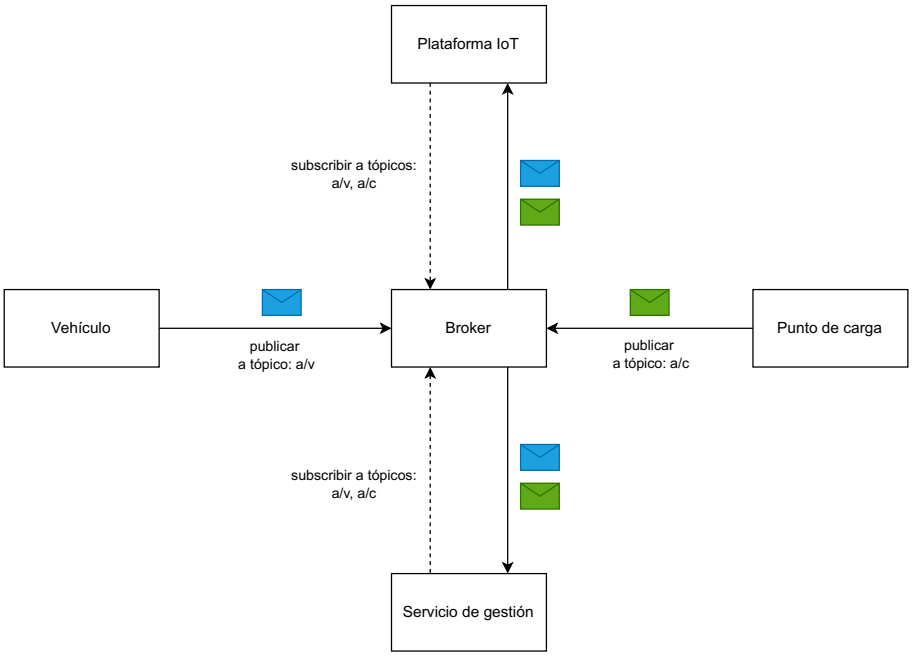


Figura 3.1: Comunicación de telemetría usando *middleware*

Si bien puede parecer que un componente más puede hacer que la comunicación sea más compleja, la realidad es que es todo lo contrario, un bróker ahorra la implementación de módulos de comunicación específicos para cada componente y a la vez abstrae la tecnología con la que se ha implementado cada uno. De este modo también se facilita en gran parte la incorporación de nuevos componentes ya sean dispositivos u otras aplicaciones para sumar funcionalidad al sistema existente.

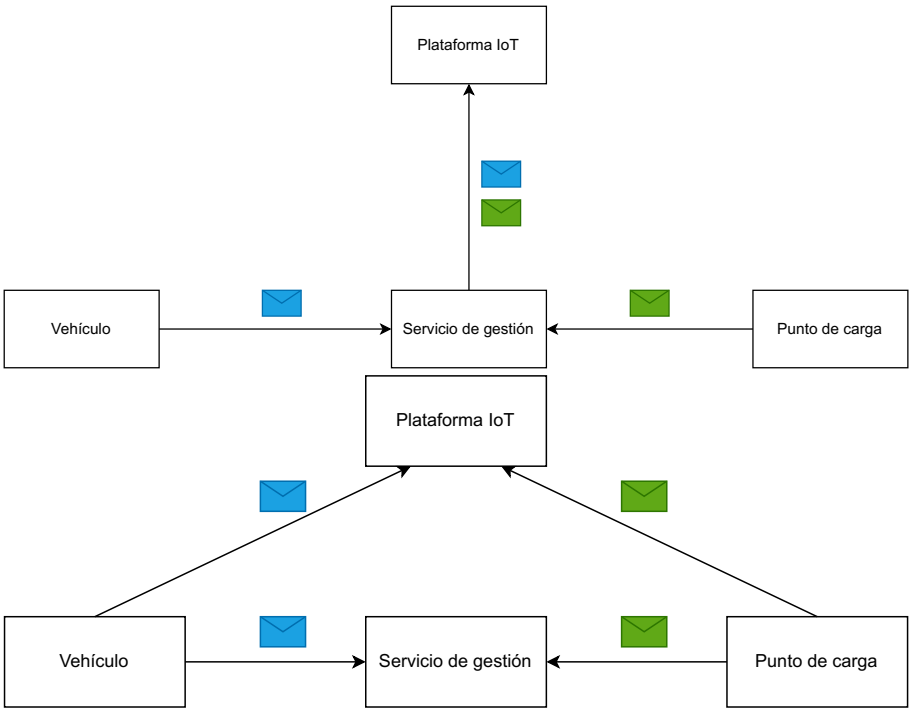


Figura 3.2: Comunicación de telemetría sin usar middleware

La figura anterior muestra dos formas alternativas de como se llevaría a cabo el envío de telemetría, ambas formas son muy ineficientes, la primera porque supondría una gran sobrecarga para el servicio de gestión en caso de disponer de un gran número de dispositivos conectados y la segunda porque a largo plazo, acabaría suponiendo un mayor coste energético para los dispositivos con autonomía limitada.

3.2.3. Plataforma IoT

Para elegir una plataforma IoT nos basaremos en diversos factores:

- Que sea de código abierto.
- Licencia de uso, debe permitir el uso comercial.
- Funcionalidades y limitaciones.

En el capítulo 2 habíamos mencionado tres plataformas, ThingsBoard y Openremote son las plataformas que menos limitaciones tienen, entre ellas el número de dispositivos que soportan, reduciéndose este número al que las capacidades del hardware en el que se despliegue la plataforma pueda manejar. También son las más completas a nivel de funcionalidad. Openremote incluye, en parte, las mismas funcionalidades que la versión de pago de ThingsBoard pero cuenta con la desventaja de que tiene una licencia de tipo AGPLv3[19] que obliga a que cualquier *software* que se combine estrechamente con Openremote quede bajo la misma licencia. Una empresa que desarrolla *software* comercial que utiliza código AGPLv3 no podrá distribuir el *software* comercial sin también distribuir el código fuente completo. Esto puede ser un inconveniente y puede exponer el código

comercial a la competencia. En cambio ThingsBoard tiene una licencia de tipo *Apache License 2.0*[18] la cual es menos restrictiva y permite la libre modificación y distribución del código.

Otros factores a tener en cuenta es el soporte que este tipo de *software* recibe de la comunidad, ThingsBoard al tener una mayor popularidad cuenta con más apoyo, además de tener una mejor documentación. Otra de las ventajas de ThingsBoard es que ofrece una mayor variedad de herramientas para la visualización de datos. Una de sus principales desventajas es que su versión gratuita no soporta directamente la integración de diferentes protocolos de comunicación IoT, sin embargo, hay un componente adicional llamado ThingsBoard Gateway que se encarga de conectar la plataforma con sistemas de terceros.

3.2.4. Base de datos

En el contexto de Internet de las Cosas (IoT), hay varios tipos de bases de datos que pueden ser utilizados para almacenar y gestionar los datos generados por los dispositivos conectados. La elección de la base de datos adecuada depende de factores como el volumen de datos, la velocidad de ingreso, los requisitos de consulta y análisis, la escalabilidad y la disponibilidad. A continuación, se presentan algunos tipos de bases de datos relevantes para IoT:

1. **Bases de Datos Relacionales:** Estas bases de datos son conocidas por su estructura tabular y la capacidad de manejar relaciones entre datos. Son adecuadas para aplicaciones que tienen un esquema de datos predefinido y requieren transacciones ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad).

2. **Bases de Datos de Series Temporales:** Estas bases de datos están optimizadas para almacenar y consultar datos de series temporales, como los generados por sensores en IoT. Son excelentes para casos de uso que involucran datos de tiempo, ya que permiten consultas y análisis rápidos de series temporales.

3. **Bases de Datos NoSQL:** Las bases de datos NoSQL son flexibles y escalables, lo que las hace adecuadas para escenarios de IoT con volúmenes variables y rápidos cambios de datos. Son ideales para datos no estructurados y semiestructurados.

4. **Bases de Datos en Memoria:**

   Estas bases de datos almacenan datos en la memoria principal, lo que permite un acceso muy rápido. Este tipo es útil para casos de uso en los que se requiere baja latencia en la recuperación de datos, pero pueden no ser ideales para almacenar grandes cantidades de datos a largo plazo.

5. **Bases de Datos Distribuidas:** Como su propio nombre indica, están diseñadas para funcionar en entornos distribuidos y escalables, lo que las hace adecuadas para escenarios de IoT en los que se requiere alta disponibilidad y escalabilidad horizontal.

6. **Bases de Datos Grafo:**

   Este tipo de está diseñado para modelar y almacenar datos con relaciones complejas. Puede ser útil en escenarios de IoT que implican la comprensión de conexiones y relaciones entre múltiples dispositivos.

3.2.5. Simulación

Como bien se mencionó al inicio, este trabajo es un prototipo funcional, es decir, una simulación. La simulación permite imitar el comportamiento de un sistema en el tiempo y en base a eventos. En la ingeniería de sistemas hay varios tipos de simulación en los sistemas distribuidos, aunque solo nos interesan dos tipos:

- **Simulación basada en agentes**: se modelan individualmente los nodos o agentes en el sistema distribuido. Cada agente tiene su propio comportamiento y reglas, pudiendo interactuar con otros agentes según ciertas condiciones. Este tipo de simulación se utilizará principalmente para imitar el comportamiento de vehículos y puntos de carga así como las interacciones entre ellos y el servicio de gestión.
- **Simulación basada en eventos discretos**: este es uno de los enfoques más comunes en sistemas distribuidos. Se modela el sistema como una serie de eventos discretos que ocurren en momentos específicos y afectan el estado del sistema. Los eventos pueden incluir la llegada de mensajes, el procesamiento de tareas, cambios de estado en los nodos, etc. La simulación sigue el tiempo paso a paso, avanzando solo cuando ocurre un evento, y registra cómo afecta al sistema distribuido en su conjunto. La simulación por pasos es un tipo derivado, cada evento o paso representa un intervalo de tiempo y las actualizaciones en el sistema ocurren en esos intervalos. Los simuladores de pasos son útiles para modelar sistemas que funcionan en etapas discretas, como sistemas de producción y procesos industriales.

3.3 Solución propuesta

Por todo lo expuesto en la sección anterior, la solución a desarrollar tendría como características:

- Comunicaciones mediante REST para transmitir mensajes importantes al servicio de gestión pero poco frecuentes.
- Uso del protocolo MQTT para el envío de telemetría, dada su ligereza y eficiencia.
- Utilización de un bróker MQTT para entregar los mensajes procedentes de la telemetría a la plataforma IoT y al servicio de gestión. Cualquiera de las opciones son válidas, pero aquellas que ofrezcan escalabilidad son preferentes en situaciones de alta carga. Se optará por EMQX ya que además ofrece diversos paneles para monitorizar el estado del bróker.

- Una base de datos NoSQL, debido a su rapidez y flexibilidad para almacenar y modificar los datos. Se han descartado las bases de datos de series temporales por dos razones: lo más importante para el servicio de gestión es el estado más reciente de cada vehículo y punto de carga y la otra razón es que las plataformas IoT suelen incluir este tipo de base de datos por defecto, generar estadísticas y análisis de esos datos no debe ser una responsabilidad del servicio de gestión, sino de la plataforma IoT. La elección tomada es MongoDB, ya que permite estructuras de datos más flexibles y soporta la transaccionalidad.
- Una plataforma IoT, esta debe tener una variedad de características, un soporte activo por parte de la comunidad e idealmente una licencia que permita el uso comercial.
- Un simulador por pasos para guiar el comportamiento y cambios de estado de cada vehículo y punto de carga simultáneamente.

CAPÍTULO 4
Diseño de solución

Una vez definidos y especificados los requisitos, se puede pasar a la fase de diseño de la solución. En este capítulo se detallará el diseño a nivel global, es decir, la arquitectura del sistema y a un nivel más detallado, interno de cada uno de los componentes a desarrollar.

4.1 Arquitectura global del sistema

A nivel de arquitectura, el sistema contará con los siguientes componentes y comunicaciones:

- Servicio de gestión: puede recibir peticiones REST, también debe poder recibir y enviar mensajes MQTT a vehículos y puntos de carga.
- Vehículos y puntos de carga: envían peticiones REST al servicio de gestión y deben poder recibir y enviar mensajes a través de MQTT.
- Bróker: se encargará de hacer llegar los mensajes MQTT a los tópicos a los que fueron enviados.
- Base de datos: guardará y actualizará el estado más reciente de vehículos y puntos de carga, también almacenará los repartos entrantes y reservas.
- Plataforma IoT: se utilizará para mostrar de manera visual el estado de cargadores y vehículos.
- Simulador por pasos: El envío de pasos es una comunicación que sucede frecuentemente, la mejor forma es enviar los pasos a través del bróker.

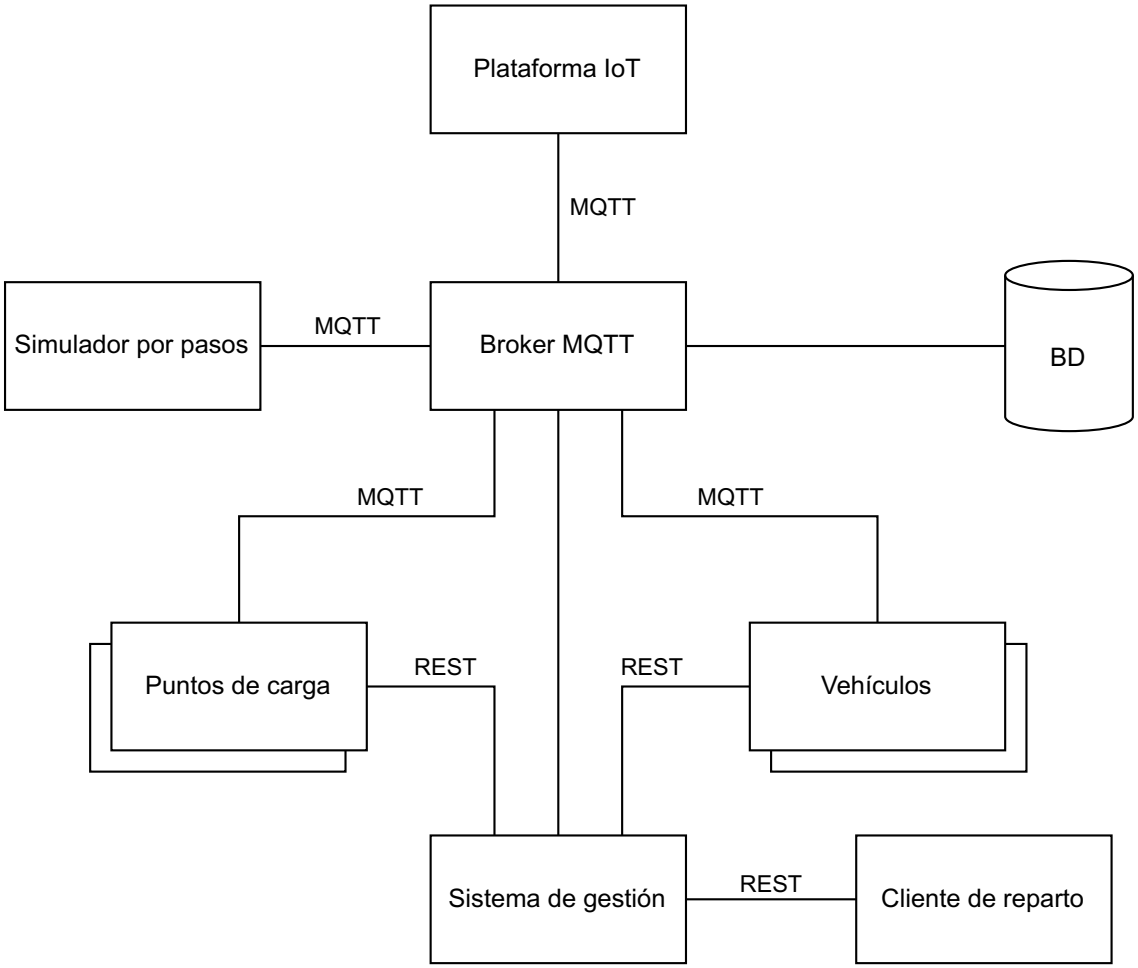


Figura 4.1: Arquitectura global del sistema

Cabe resaltar que las comunicaciones representadas en realidad son más complejas y se llevan a cabo en diferentes situaciones, por tanto se debe ampliar el diagrama 4.1 en función de los requisitos definidos en el capítulo anterior. Existen componentes que tienen casos de uso relacionados o en su defecto, muy similares, por este motivo en la sección siguiente se agruparán algunos de ellos y se refinará el diseño de las comunicaciones.

4.2 Diseño detallado de las comunicaciones en los casos de uso

4.2.1. Simulador por pasos

Empezaremos por el simulador, esta pieza es fundamental para un comportamiento adecuado de los vehículos, puntos de carga y en cierta medida para el servicio de gestión. El simulador implementará un contador de pasos que se incrementará con cada envío realizado al bróker.

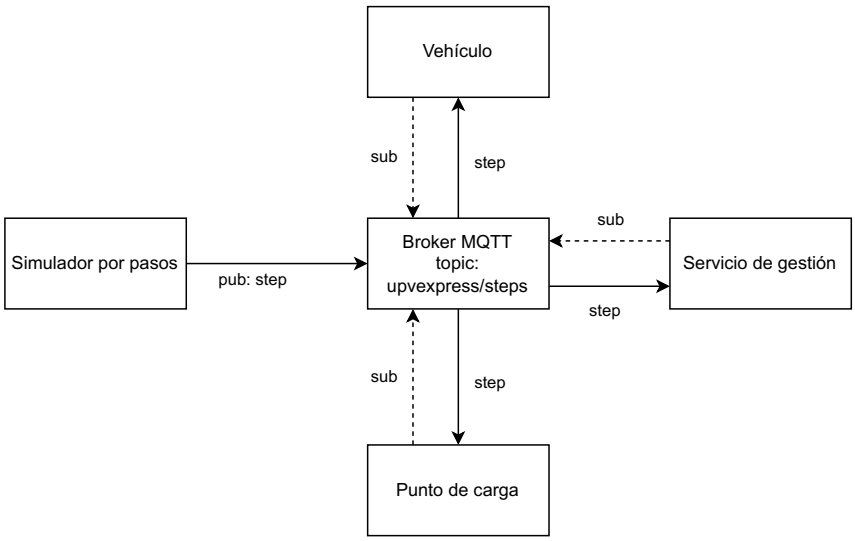


Figura 4.2: Comunicaciones MQTT para el simulador

En MQTT todos los mensajes se envían a un tópico determinado al cual los interesados se pueden suscribir. El tópico puede tener cualquier nombre mientras siga las reglas del protocolo, en nuestro caso el tópico será la combinación del nombre de una empresa ficticia, llamemosla *upvexpress*, y un nombre relevante al caso de uso, en este caso *steps*, *upvexpress/steps*. Todos los elementos que se muestran en la figura 4.2 deben suscribirse al tópico al iniciar. Una vez suscritos, con cada paso los vehículos deberán moverse en caso de tener un reparto asignado, por su parte los puntos de carga deberán suministrar esperar a un vehículo en caso de tener una reserva y suministrar carga si el vehículo esperado ha llegado.

4.2.2. Registrar vehículos y puntos de carga

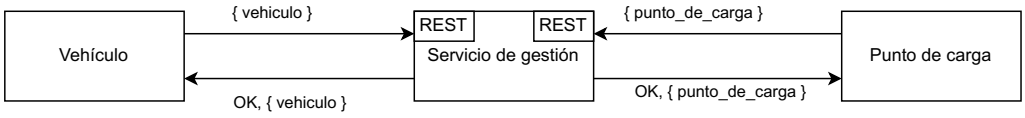


Figura 4.3: Comunicaciones REST para el registro

Este caso de uso ocurre cuando cualquier vehículo o punto de carga inicie, tal y como se muestra en la figura 4.3 realizarán una petición REST de tipo POST al servicio de gestión enviando un objeto JSON con su identificador y posición al momento de iniciar. En caso de que el registro sea exitoso el servicio responderá con un OK y el mismo objeto que se ha enviado. En caso de que el objeto enviado sea incorrecto, el servicio debe responder con BAD_REQUEST.

4.2.3. Solicitar reparto

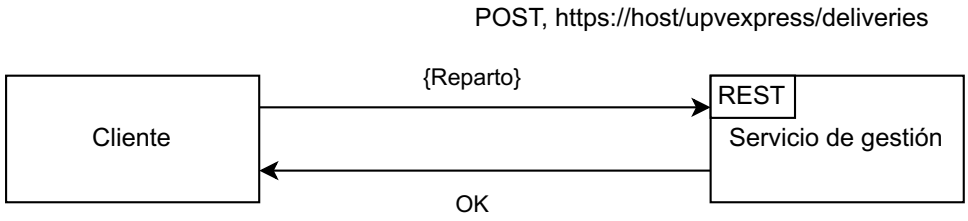


Figura 4.4: Comunicaciones REST para solicitar un reparto

Tal y como se muestra en la figura anterior, el cliente realiza un petición REST de tipo POST enviando un objeto JSON del reparto(con las posiciones de recogida y entrega), si la solicitud es válida el servicio responderá con un OK y el mismo reparto pero con un identificador añadido. Si la solicitud es incorrecta, el servicio debe responder con BAD_REQUEST.

4.2.4. Enviar estado

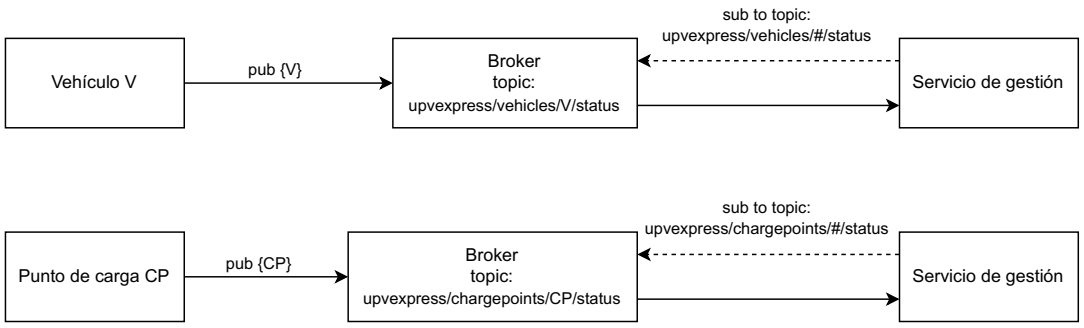


Figura 4.5: Envío del estado de vehículos y puntos de carga a través de MQTT

Tanto vehículos como puntos de carga deben enviar periódicamente su estado al servicio de gestión, esto es, con cada paso que sea enviado por el simulador. MQTT permite la suscripción a tópicos utilizando *wildcards* o también llamados comodines siendo estos "#", que permite la suscripción a tópicos y subtópicos de todos los niveles, y "+", que permite la suscripción a tópicos y subtópicos del mismo nivel solamente. En la figura 4.5 se muestra como el servicio de gestión realiza una suscripción utilizando comodines para recibir el estado de todos los vehículos y cargadores posibles, esto garantiza que aunque se registren vehículos nuevos, el servicio estará preparado para recibir sus actualizaciones de estado.

4.2.5. Asignar repartos

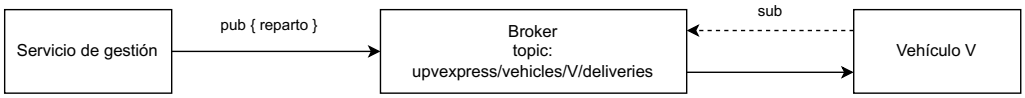


Figura 4.6: Envío del reparto asignado a través de MQTT

Con cada paso del simulador, el servicio intentará asignar los repartos pendientes a los vehículos disponibles. Los vehículos se suscribirán a un tópico específico, como el de la figura 4.6 por donde les llegarán los repartos. Una vez el vehículo haya aceptado el reparto iniciará su marcha hacia la posición de recogida.

4.2.6. Enviar estado de reparto

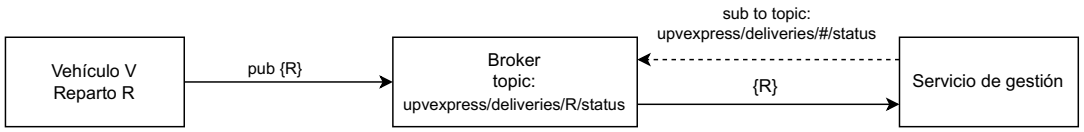


Figura 4.7: Envío del estado de reparto a través de MQTT
??

Una vez que un vehículo ha recibido un reparto, al igual que sucede con su propio estado, deberá enviar el estado de su reparto a un tópico designado específicamente para ello. El servicio de gestión recibirá todas las actualizaciones excepto la última, cuando se ha entregado el reparto, dado que ese caso es más especial. En la imagen **??** se aprecia como el servicio se vuelve a suscribir utilizando comodines.

4.2.7. Confirmar entrega de reparto

Figura 4.8: Confirmación de entrega de un reparto a través de una llamada REST

Aunque un vehículo envía el estado del reparto con cada paso del simulador, es muy importante asegurarse de que el reparto se marca como completado ya que eso implica que el vehículo que lo tenía asignado vuelve a quedar disponible. Como se puede apreciar en la figura 4.8 el vehículo identifica el reparto a completar enviando un objeto con el identificador.

4.2.8. Reservar punto de carga

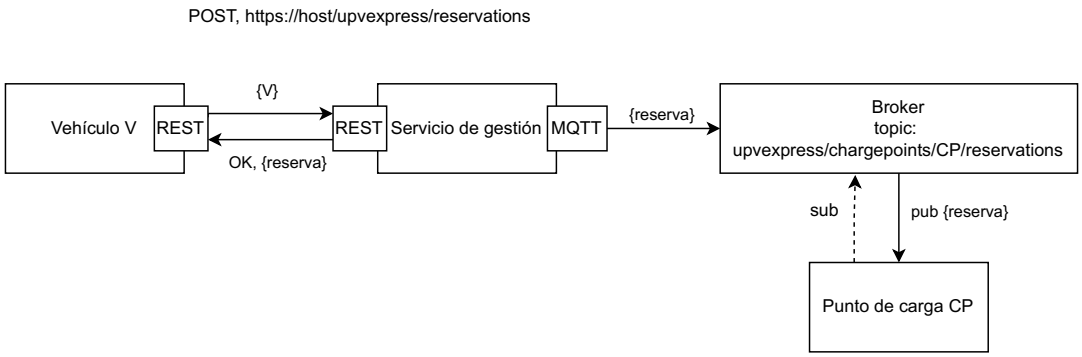


Figura 4.9: Reserva de un punto de carga

Cuando un vehículo baja de cierto porcentaje de batería debe solicitar la reserva de un punto de carga al servicio de gestión, por lo que en este caso una llamada REST es ideal. El servicio después de comprobar la disponibilidad de los cargadores, seleccionará el más cercano al vehículo y le hará llegar la reserva a través de un mensaje MQTT, tal y como se muestra en la figura 4.9. Todos los puntos de carga deben suscribirse a su tópico designado al iniciar, por donde recibirán reservas a lo largo del tiempo. En cuanto el vehículo reciba su reserva pausará su reparto y se dirigirá al punto de carga. Cuando un punto de carga recibe una reserva debe suscribirse a un tópico creado en base a la reserva donde el vehículo le indicará el inicio y fin del suministro.

4.2.9. Completar reserva

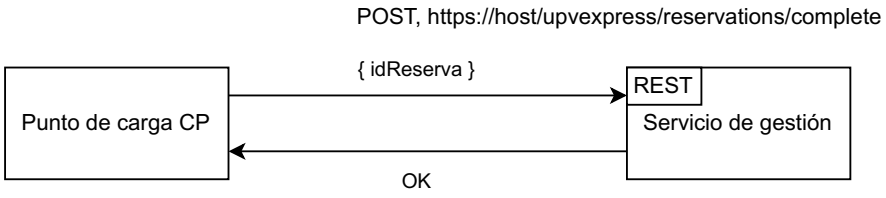


Figura 4.10: Completar reserva de un punto de carga mediante una petición REST

Como se muestra en la figura anterior, el cargador debe notificar al servicio de gestión para que este tenga en cuenta su disponibilidad en el momento que el vehículo termine su carga a través de una petición REST.

4.2.10. Iniciar carga

Cuando el vehículo que solicitó la reserva llega a la posición del punto de carga, como en la figura 4.11, este le mandará un mensaje MQTT para iniciar la carga. Una vez el vehículo indique el inicio de la carga, se pasará a la fase de suministro.

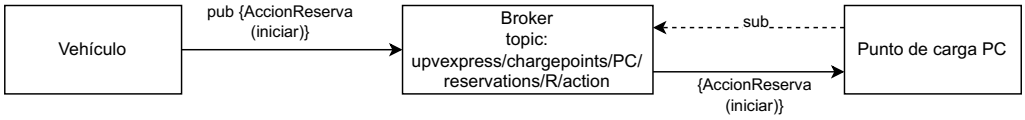


Figura 4.11: Comunicación MQTT para iniciar la carga

4.2.11. Suministrar carga

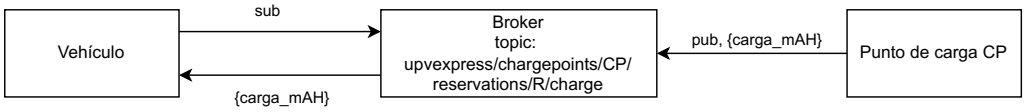


Figura 4.12: Comunicación MQTT para iniciar la carga

El punto de carga enviará un mensaje MQTT al tópico de carga asignado a la reserva con una cantidad de miliamperios, determinada en base su capacidad, con cada paso del simulador, por tanto la comunicación de la imagen 4.12 es periódica. El vehículo sumará la cantidad recibida a su capacidad restante hasta que su porcentaje de batería esté cerca del cien por cien.

4.2.12. Finalizar carga

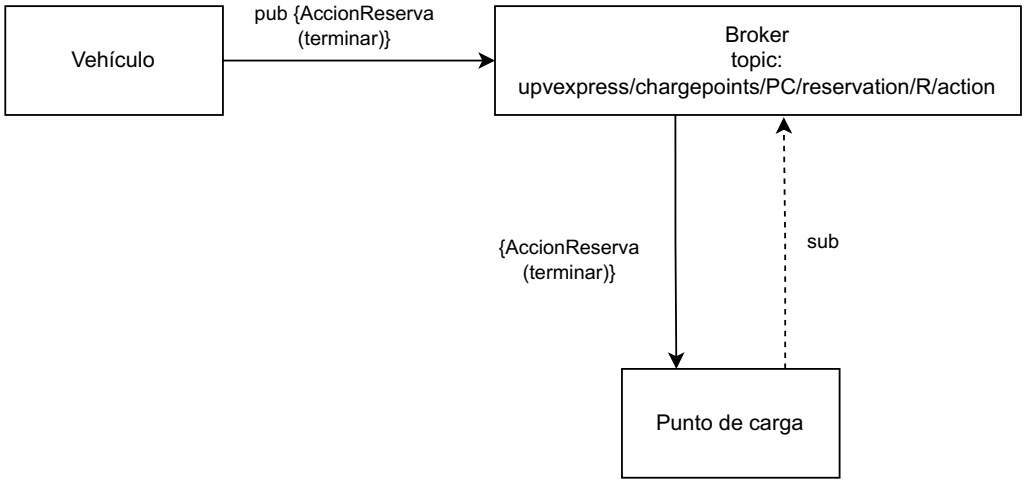


Figura 4.13: Comunicación MQTT para finalizar la carga

Una vez cargado, el vehículo debe enviar un mensaje que indique el fin de la carga y, aunque no se represente en la figura anterior, procede a anular su suscripción al tópico donde recibía los mensaje con la capacidad a cargar. El punto de carga también anulará su suscripción al tópico donde recibía las notificaciones de inicio y fin de carga.

4.3 Arquitectura Hexagonal

Para la implementación de los microservicios vehículo y punto de carga se ha elegido la arquitectura hexagonal[3][2], también conocida como arquitectura de

puertos y adaptadores, es un patrón de diseño arquitectónico propuesto por Alistair Cockburn en 2005. El objetivo de la arquitectura hexagonal es proporcionar un diseño modular y flexible que facilite el desarrollo, la prueba y el mantenimiento de aplicaciones.

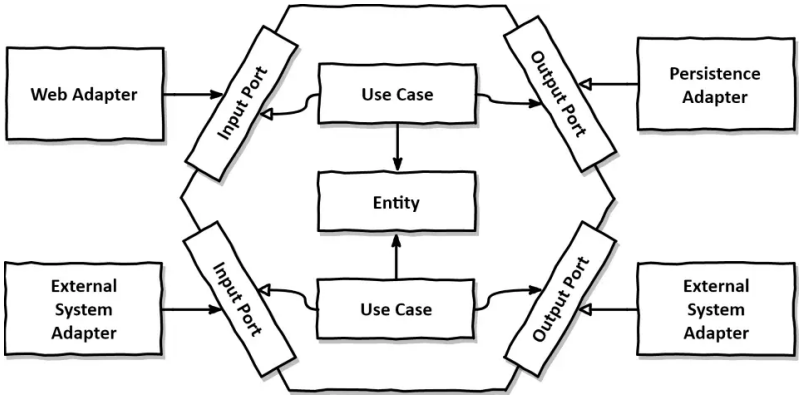


Figura 4.14: Arquitectura hexagonal. Fuente: https://reflectoring.io/spring-hexagonal/

La arquitectura se basa en la idea de que el núcleo de la aplicación, que contiene el modelo de dominio y los servicios de aplicación, debe estar separado de las dependencias externas, como la interfaz de usuario, la base de datos y los servicios de terceros. Este desacoplamiento se logra mediante el uso de puertos y adaptadores.

Los puertos son interfaces abstractas que proporcionan puntos de entrada y salida para el núcleo de la aplicación. Los adaptadores son implementaciones concretas de los puertos que proporcionan la conexión entre el núcleo de la aplicación y las dependencias externas.

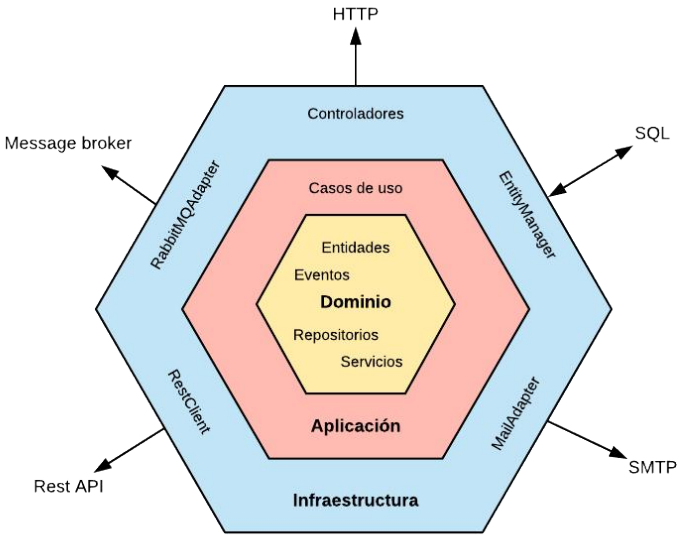


Figura 4.15: Arquitectura hexagonal. Fuente: https://medium.com/@edusalguero/arquitectura-hexagonal-59834bb44b7f

La arquitectura hexagonal se puede representar como un hexágono, con el núcleo de la aplicación en el centro y los puertos y adaptadores en los bordes. Cada lado del hexágono representa un puerto concreto, aunque en la práctica podrían haber más puertos distintos junto con su correspondiente adaptador.

Existen distintas interpretaciones de esta arquitectura, algunas al igual que en la figura 4.15 tienen los servicios y puertos en la capa de dominio mientras que otras sitúan esos elementos en la capa de aplicación, dejando en la capa de dominio solamente las entidades y eventos.

La arquitectura hexagonal proporciona una serie de beneficios, entre los que se incluyen:

- Facilita el desarrollo, la prueba y el mantenimiento de aplicaciones. Al separar el núcleo de la aplicación de las dependencias externas, se puede desarrollar y probar el núcleo de forma aislada. Esto facilita el mantenimiento de la aplicación cuando se modifican las dependencias externas.

- Mejora la flexibilidad y la escalabilidad de las aplicaciones. Al utilizar puertos y adaptadores, se puede cambiar fácilmente las dependencias externas de la aplicación. Esto permite a las aplicaciones adaptarse a nuevos requisitos o entornos.

- Facilita la integración con sistemas externos. Al utilizar puertos y adaptadores, se puede conectar fácilmente las aplicaciones con otros sistemas externos. Esto permite a las aplicaciones intercambiar datos e información con otros sistemas.

Las desventajas de esta arquitectura son:

- Su rápido crecimiento en cuanto a número de clases, ya que cada caso de uso requiere sus propias interfaces e implementaciones. Este problema de crecimiento puede resolverse diviendiendo la aplicación en varios contextos, cada uno con su cada de dominio, aplicación e infraestructura.

- El tiempo necesario para implementar la estructura inicial, ya que cada caso de uso puede necesitar diferentes necesidades en cuanto a puertos, se invierte mucho tiempo creando esas interfaces y toda la estructura de carpetas.

4.4 Diseño detallado de los componentes

En esta sección se detallarán las clases de los componentes con mayor complejidad. Estos componentes son desarrollados siguiendo la arquitectura descrita anteriormente. Las clases definidas para cada se sitúan en la capa de dominio. Por convención todos los nombres de las clases que aparezcan las figuras de esta sección serán inglés.

4.4.1. Servicio de gestión

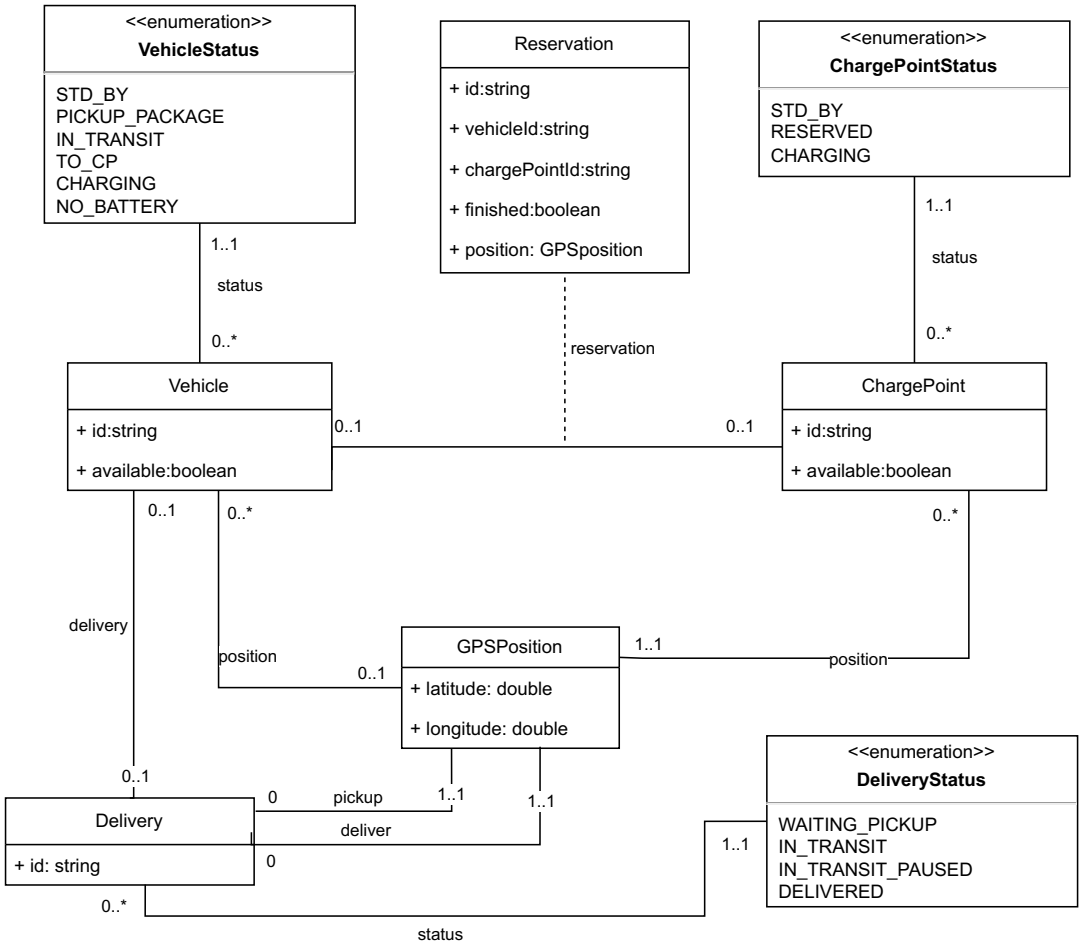


Figura 4.16: Diagrama de clases del servicio de gestión

El diagrama anterior muestra que las clases del servicio son las siguientes:

- *Vehicle:* Esta clase como su nombre indica representa un vehículo.
  - *id*: Identificador.
  - *available*: marca la disponibilidad de un vehículo.
  - *status*: Todo vehículo tiene un estado asociado en cada momento, por defecto *STD_BY*.
  - *delivery*: Puede haber o no un reparto asignado.
  - *position*: Posición actual del vehículo, usada para determinar el punto de carga y reparto más cercanos.
  - *reservation*: Reserva asociada, puede haber como mucho una reserva en cada momento.
- *VehicleStatus:* Enumerado con los posibles estados de un vehículo, sirve para ilustrar mejor su situación actual.
  - *STD_BY*: Parado.

  - *PICKUP_PACKAGE*: De camino a recoger el paquete.
  - *IN_TRANSIT*: En tránsito, de camino a la posición de entrega.
  - *TO_CP*: En camino a la posición del cargador reservado.
  - *CHARGING*: Cargando.
  - *NO_BATTERY*: Sin batería, el vehículo queda inutilizable.

- *Delivery:* Representa un reparto.
  - *pickup*: Posición de recogida del reparto.
  - *deliver*: Posición de entrega.
  - *status*: Todo reparto tiene un estado asociado en cada momento, por defecto *WAITING_PICKUP*.

- *DeliveryStatus:* Enumerado con los posibles estados de un reparto.
  - *WAITING_PICKUP*: Esperando recogida.
  - *IN_TRANSIT*: En tránsito.
  - *IN_TRANSIT_PAUSED*: Tránsito en espera, aplicable cuando un vehículo debe dirigirse a un punto de carga.
  - *DELIVERED*: Entregado.

- *GPSPosition:* Posición GPS empleada por vehículos y puntos de carga
  - *Latitude*: Latitud.
  - *Longitude*: Longitud.

- *ChargePoint:* Enumerado con los posibles estados de un punto de carga.
  - *id*: Identificador.
  - *status*: Todo punto de carga tiene un estado asociado en cada momento, por defecto *STD_BY*.
  - *position*: Posición actual del cargador, usada para la determinación de una reserva.
  - *reservation*: Reserva asociada, puede haber como mucho una reserva en cada momento.

- *ChargePointStatus:* Enumerado con los posibles estados de un punto de carga, sirve para ilustrar mejor su situación actual.
  - *STD_BY*: Parado.
  - *RESERVED*: Reservado por un vehículo.
  - *CHARGING*: Cargando un vehículo.

- *Reservation:* Reserva, resultado de la solicitud de un vehículo.
  - *id*: Identificador de la reserva.
  - *vehicleId*: Identificador del vehículo.

- *chargePointId*: Identificador del punto de carga.
- *finished*: marca la finalización, concretamenta la terminación de la carga.
- *position*: Posición del punto de carga.

4.4.2. Vehículo

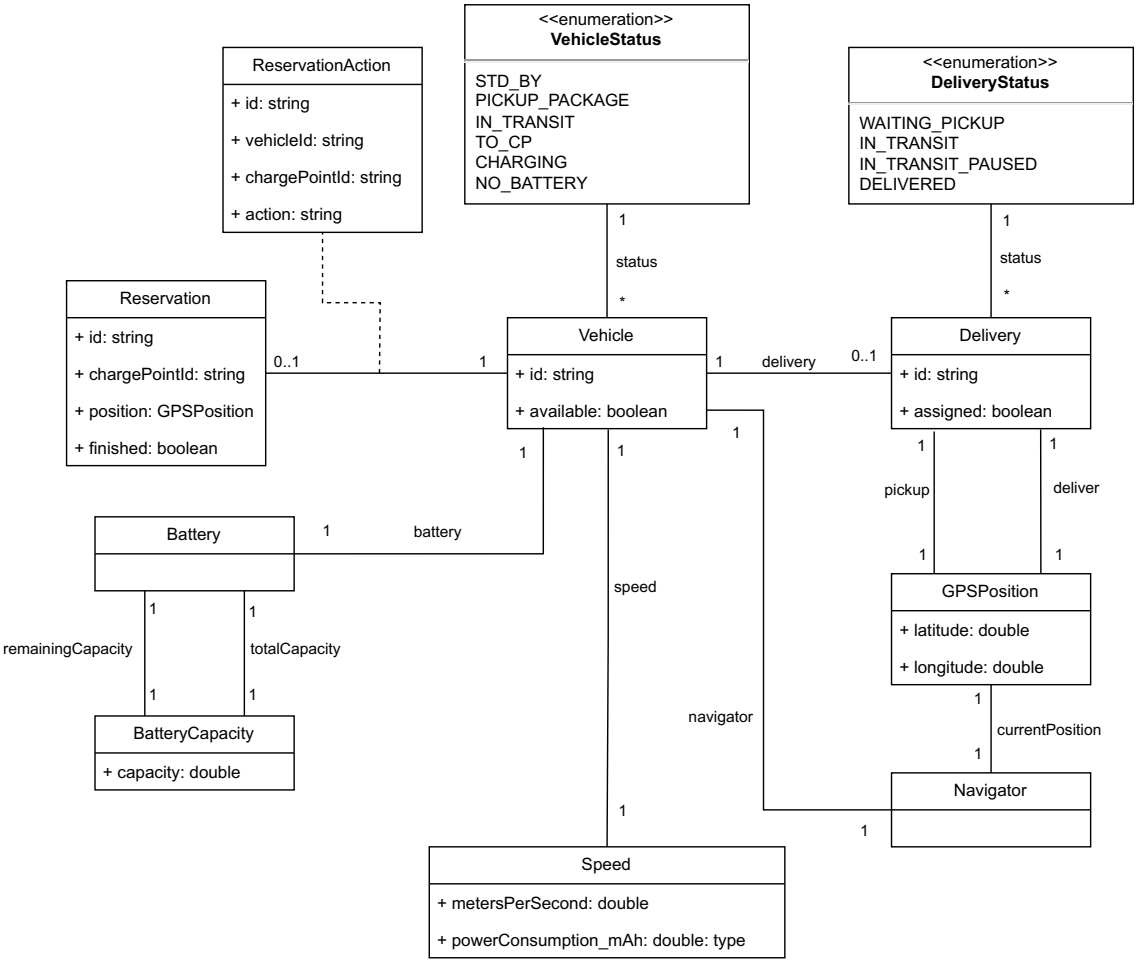


Figura 4.17: Diagrama de clases del microservicio vehículo

Como se muestra en el diagrama anterior, las clases del microservicio vehículo son:

- *Vehicle:* Esta clase como su nombre indica representa un vehículo.
  - *id*: Identificador.
  - *available*: marca la disponibilidad de un vehículo.
  - *status*: Todo vehículo tiene un estado asociado en cada momento, por defecto *STD_BY*.
  - *delivery*: Puede haber o no un reparto asignado.
  - *navigator*: navegador, siempre asociado a un vehículo.

  - *reservation*: Reserva asociada, puede haber como mucho una reserva en cada momento.

- *VehicleStatus:* Enumerado con los posibles estados de un vehículo, sirve para ilustrar mejor su situación actual.
  - *STD_BY*: Parado.
  - *PICKUP_PACKAGE*: De camino a recoger el paquete.
  - *IN_TRANSIT*: En tránsito, de camino a la posición de entrega.
  - *TO_CP*: En camino a la posición del cargador reservado.
  - *CHARGING*: Cargando.
  - *NO_BATTERY*: Sin batería, el vehículo queda inutilizable.
- *Navigator:* Navegador GPS del vehículo, dicta el rumbo a seguir.
  - *position*: Posición actual del vehículo.
- *Speed:* Velocidad del vehículo .
  - *metersPerSecond*: Velocidad expresada en metros por segundo.
  - *powerConsumption_mAh*: Consumo eléctrico a una determinada velocidad.
- *Delivery:* Representa un reparto.
  - *pickup*: Posición de recogida del reparto.
  - *deliver*: Posición de entrega.
  - *status*: Todo reparto tiene un estado asociado en cada momento, por defecto *WAITING_PICKUP*.
- *DeliveryStatus:* Enumerado con los posibles estados de un reparto.
  - *WAITING_PICKUP*: Esperando recogida.
  - *IN_TRANSIT*: En tránsito.
  - *IN_TRANSIT_PAUSED*: Tránsito en espera, aplicable cuando un vehículo debe dirigirse a un punto de carga.
  - *DELIVERED*: Entregado.
- *GPSPosition:* Posición GPS empleada por el vehículo.
  - *Latitude*: Latitud.
  - *Longitude*: Longitud.
- *Battery:* Batería del vehículo.
  - *remainingCapacity*: Capacidad restante.
  - *totalCapacity*: Capacidad máxima de la batería.
- *BatteryCapacity:* Capacidad de la bateria.

- *capacity*: Valor de la capacidad en mAH.

- *Reservation:* Reserva, resultado de la solicitud de un vehículo.
  - *id*: Identificador de la reserva.
  - *chargePointId*: Identificador del punto de carga.
  - *finished*: marca la finalización, concretamente la terminación de la carga.
  - *position*: Posición del punto de carga.
- *ReservationAction:* Acción a ejecutar sobre una reserva, indica al cargador si iniciar o terminar la carga.
  - *id*: Identificador de la reserva.
  - *vehicleId*: Identificador del vehículo.
  - *chargePointId*: Identificador del punto de carga.
  - *action*: el propio valor de la acción.

4.4.3. Punto de carga

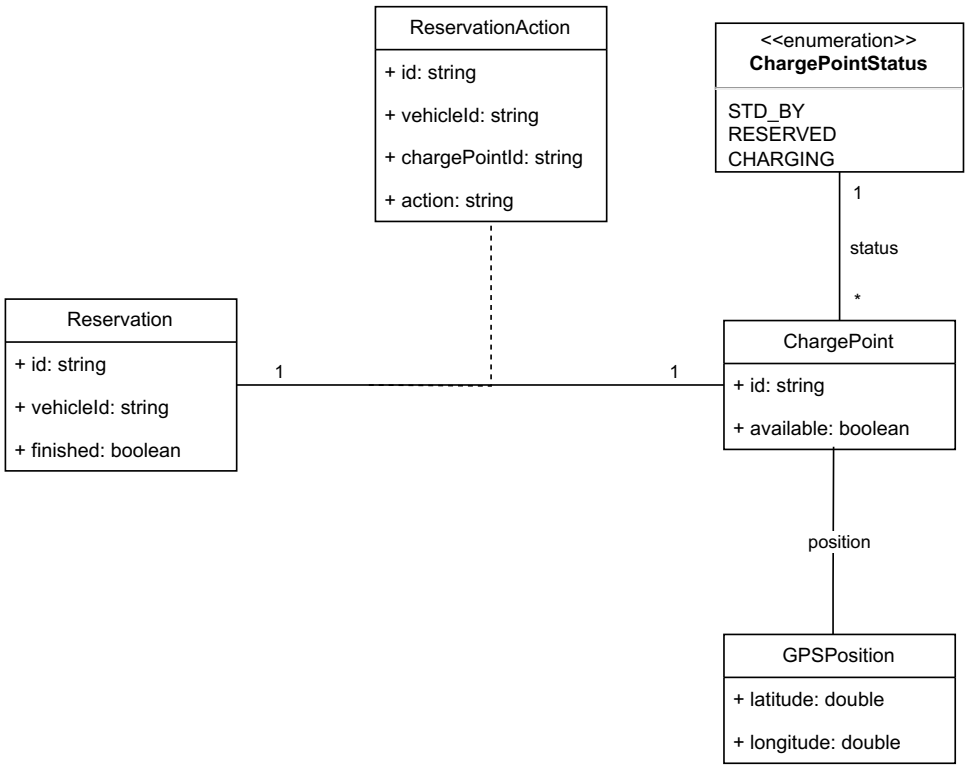


Figura 4.18: Diagrama de clases del microservicio punto de carga

Observando la figura anterior se puede apreciar que en este caso las clases son:

- *ChargePoint:* Representa un punto de carga.

  - *id*: Identificador.
  - *status*: Todo punto de carga tiene un estado asociado en cada momento, por defecto *STD_BY*.
  - *position*: Posición actual del cargador, usada para la determinación de una reserva.
  - *reservation*: Reserva asociada, puede haber como mucho una reserva en cada momento.

- *ChargePointStatus:* Enumerado con los posibles estados de un punto de carga, sirve para ilustrar mejor su situación actual.
  - *STD_BY*: Parado.
  - *RESERVED*: Reservado por un vehículo.
  - *CHARGING*: Cargando un vehículo.

- *Reservation:* Reserva, resultado de la solicitud de un vehículo.
  - *id*: Identificador de la reserva.
  - *vehicleId*: Identificador del vehículo.
  - *finished*: marca la finalización, concretamente la terminación de la carga.

- *ReservationAction:* Acción a ejecutar sobre una reserva, indica al cargador si iniciar o terminar la carga.
  - *id*: Identificador de la reserva.
  - *vehicleId*: Identificador del vehículo.
  - *chargePointId*: Identificador del punto de carga.
  - *action*: el propio valor de la acción.

- *GPSPosition:* Posición GPS empleada por el punto de carga
  - *Latitude*: Latitud.
  - *Longitude*: Longitud.

CAPÍTULO 5
Tecnología utilizada

5.1 Opciones para el middleware de mensajería

En el capítulo 3 se ha expuesto la funcionalidad adicional que brindan los middlewares de mensajería o brókers y por tanto la necesidad de elegir uno. Algunos de los más populares actualmente son:

1. **Mosquitto:**
   - **Descripción:** Mosquitto[20] es un bróker MQTT de código abierto desarrollado por la Fundación Eclipse. Es conocido por su ligereza y facilidad de configuración, lo que lo hace adecuado para aplicaciones pequeñas y medianas. Ofrece funciones de autenticación y encriptación.
   - **Características clave:**
     - Soporte para MQTT versión 3.1 y 3.1.1.
     - Autenticación y control de acceso.
     - Transporte SSL/TLS para encriptación.
     - Funciones de logging y depuración.
2. **HiveMQ:**
   - **Descripción:** HiveMQ[22] es un bróker MQTT escalable que ofrece una edición de código abierto, así como una edición comercial con características avanzadas. Es conocido por su capacidad de gestión de alto rendimiento y alta disponibilidad.
   - **Características clave:**
     - Soporte para MQTT versión 3.1, 3.1.1 y 5.0.
     - Escalabilidad horizontal para manejar cargas de trabajo grandes.
     - Integración con tecnologías de nube y microservicios.
     - Monitoreo y control centralizados.
3. **VerneMQ:**
   - **Descripción:** VerneMQ[21] es un bróker MQTT distribuido y escalable de código abierto. Está diseñado para ofrecer alta disponibilidad y tolerancia a fallos, lo que lo hace adecuado para aplicaciones críticas.

- **Características clave:**
  - Escalabilidad con partición y replicación de datos.
  - Tolerancia a fallos y recuperación automática.
  - Compatibilidad con MQTT versión 3.1 y 3.1.1.
  - Encriptación SSL/TLS y autenticación.

4. **EMQX:**
   - **Descripción:** EMQX[23] es un bróker MQTT y MQTT-SN (MQTT para redes inalámbricas) de código abierto. Destaca por su alta capacidad de concurrencia y escalabilidad, siendo capaz de manejar grandes volúmenes de conexiones y mensajes.
   - **Características clave:**
     - Soporte para MQTT versión 3.1, 3.1.1 y 5.0, así como MQTT-SN.
     - Módulos de extensión para integración con sistemas externos.
     - Escalabilidad mediante múltiples instancias y clusters.
     - Autenticación, autorización y encriptación.

5.2 Tecnologías de desarrollo

5.2.1. Spring Boot

Spring Boot[24] es un módulo desarrollado como una extensión del *framework Spring*[25] con el fin de facilitar la creación de microservicios y aplicaciones web. Una de sus bondades y principal diferencia con *Spring* reside en la convención sobre configuración[], que trata de minimizar las decisiones de los desarrolladores, teniendo éstos que especificar solamente los aspectos poco convencionales. Otros aspectos muy positivos de este *framework* son la inversión de control[] y la inversión de dependencias[], que pueden ayudar a reducir en buena medida la cantidad de código a escribir.

5.2.2. Docker

Docker[28] es una plataforma para el desarrollo, distribución y ejecución de aplicaciones. Esta plataforma se utiliza para empaquetar *software* en contenedores(similares a una máquina virtual pero más ligeros)[29], proporcionando un entorno que contenga todo lo necesario para su ejecución. Esto permite ejecutar el *software* en prácticamente la totalidad sin necesidad de realizar ninguna configuración más que la propia instalación de *Docker*. Esta plataforma cuenta con su propio repositorio en línea llamado *Dockerhub* para facilitar la distribución de imágenes.

5.2.3. Docker Compose

Docker Compose[30] es una herramienta que permite la ejecución de aplicaciones basadas en múltiples contenedores. La ejecución de esta herramienta requie-

re de un fichero donde se especifican todos los contenedores a desplegar. Esta herramienta es especialmente útil en entornos de desarrollo y pruebas dada su sencillez.

5.2.4. Docker Swarm

Docker Swarm[31] es una herramienta sencilla para la orquestación de contenedores en un *cluster* formado por múltiples nodos, ya sean máquinas reales o virtuales. Entre sus ventajas destacan la alta disponibilidad, balanceo de carga, descentralización y escalabilidad.

5.2.5. Kubernetes

Kubernetes[32] es un *software* de código abierto para la orquestación de contenedores a gran escala. Los contenedores son agrupados en *pods*, colecciones de uno o más contenedores, que son la mínima unidad lógica gestionada por Kubernetes. Este orquestador programa los *pods* para que se ejecuten en los nodos de un clúster según los recursos disponibles y los requisitos de cada contenedor. También administra automáticamente la detección de servicios, equilibrio de carga, asignación de recursos, escalado y vivacidad de los contenedores dentro de los *pods*.

5.2.6. ThingsBoard

ThingsBoard[9] es una plataforma IoT de código abierto para recolección, procesamiento, análisis y visualización de datos, que además permite la gestión de dispositivos. Soporta un gran número de protocolos actuales por lo que esta plataforma es compatible con muchísimos dispositivos. Está disponibles en dos versiones, una basada en microservicios y otro monolítica. Dispone de un gran número de componentes predefinidos para facilitar la creación de paneles donde se pondrán visualizar diferentes datos de los dispositivos en tiempo real.

5.2.7. ThingsBoard Gateway

ThingsBoard Gateway[33] es una puerta de enlace que ayuda a integrar la plataforma IoT con dispositivos conectados a sistemas de terceros o legados. Al igual que la propia plataforma, ofrece conectividad con muchos de los protocolos actuales: MQTT, REST, CAN, Modbus entre otros.

5.2.8. EMQX

EMQX[23] es un *bróker* MQTT de código abierto con un motor de procesamiento de mensajes en tiempo real de alto rendimiento, que alimenta el flujo de eventos para dispositivos IoT a escala masiva. Proporciona una amplia gama de funciones para la recopilación, el procesamiento, la visualización y la gestión de datos de dispositivos. Algunas de sus principales características son:

- Escalabilidad: puede soportar hasta 100 millones de conexiones MQTT simultáneas con una sola instancia.
- Rendimiento: puede procesar hasta 5 millones de mensajes MQTT por segundo.
- Baja latencia: garantiza una latencia de 1 a 5 milisegundos para la transmisión de datos.
- Cumplimiento de estándares: cumple con los estándares MQTT 3.1.1 y MQTT 5.0.
- Seguridad: proporciona una amplia gama de funciones de seguridad para proteger los datos de dispositivos.
- Soporte para protocolos de comunicación: admite una variedad de protocolos de comunicación para la recopilación de datos de dispositivos, incluyendo MQTT, HTTP, CoAP, AMQP y WebSockets.
- Soporte para reglas de IoT: proporciona un motor de reglas para el procesamiento y la visualización de datos de dispositivos.
- Soporte para visualización de datos: proporciona una variedad de paneles para la visualización de datos.

5.2.9. MongoDB

MongoDB[34] es una base de datos NoSQL de código abierto que almacena los datos en forma de documentos, similares al formato JSON, con un esquema dinámico. Algunas de sus ventajas son el alto rendimiento, alta disponibilidad, escalabilidad y flexibilidad. Es actualmente considerada una de las mejores opciones para analíticas de datos IoT.

5.2.10. Mongo Express

Mongo Express[35] es una interfaz web de código abierto que se utiliza para administrar bases de datos MongoDB de manera fácil y conveniente. Permite crear y borrar bases de datos, realizar las operaciones CRUD (Create, Read, Update, Delete), consultar y filtrar datos entre otras funciones. Está disponible como un proyecto Node así como un contenedor Docker.

5.3 Herramientas utilizadas

5.3.1. Visual Studio Code

Visual Studio Code[36] es un editor de código fuente desarrollado por Microsoft que ofrece una gran experiencia de desarrollo para diversos lenguajes y tecnologías. Este editor es gratuito y de código abierto, y cuenta con soprte para Windows, Mac y Linux.

5.3.2. Postman

Postman[37] es una herramienta de código abierto para desarrolladores de API que ayuda a enviar, recibir y analizar solicitudes HTTP. Es una de las herramientas más populares para la depuración de APIs, algunas de sus principales funciones son la creación de colecciones de peticiones HTTP y la captura y análisis de respuestas.

CAPÍTULO 6
Desarrollo de la solución propuesta

6.1 Arquitectura global adaptada

En el capítulo anterior se ha seleccionado como plataforma IoT dado que esta plataforma necesita de su Gateway para integrarse con otros sistemas en su versión gratuita, se ha refinado ligeramente la arquitectura mostrada en la figura 4.1.

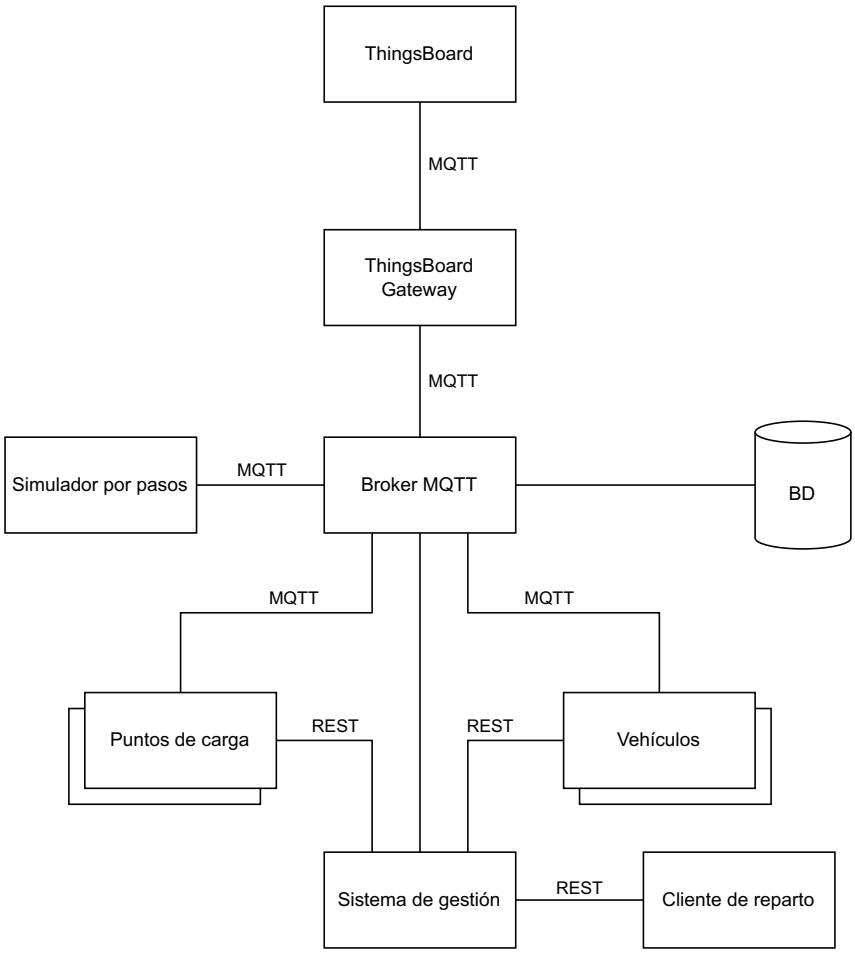


Figura 6.1: Arquitectura adaptada a la plataforma IoT

En la figura anterior se puede apreciar que el elemento Plataforma IoT se ha dividido en dos elementos, pero a nivel general se siguen pudiendo considerar como un solo elemento.

6.2 Configuración de un proyecto en SpringBoot

```
<dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-integration</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-mongodb</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-rest</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.eclipse.paho</groupId>
            <artifactId>org.eclipse.paho.client.mqttv3</artifactId>
            <version>1.2.5</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springdoc</groupId>
            <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
            <version>2.1.0</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.integration</groupId>
            <artifactId>spring-integration-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

Figura 6.2: Sección de dependencias del archivo «pom.xml»

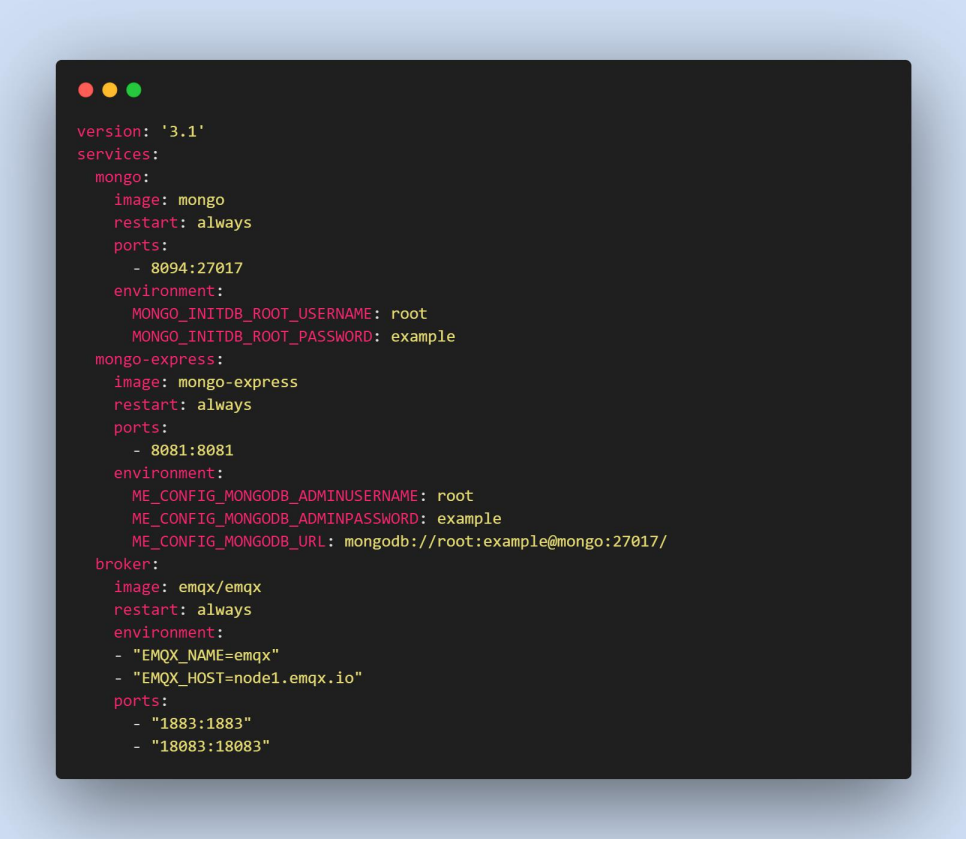
La manera más sencilla de crear un proyecto de Spring Boot es acceder a la herramienta Spring Initializr[26] donde se muestran diferentes opciones como el gestor de dependencias, lenguaje, versión, formato de empaquetamiento y algunas dependencias iniciales. Una vez configurado el proyecto se genera un archivo comprimido que se deberá extraer e importar en el entorno de desarrollo. A continuación deberemos añadir las dependencias necesarias, como se ha elegido

Maven[27] como gestor de dependencias, tendremos en la raíz de proyecto un archivo llamado «pom.xml» donde se añadirán dichas dependencias. las dependencias instaladas se podrán observar en la figura 6.2.

6.3 Despliegue de MongoDB y EMQX

Para la fase de desarrollo se ha optado por utilizar Docker Compose para el despliegue de la base de datos y el bróker MQTT, dado que es una herramienta muy sencilla y fácil de utilizar que nos ahorra la instalación de estos componentes. Adicionalmente se añadirá el contenedor "mongo-express" para poder visualizar el estado de la base de datos y realizar alguna prueba.

Para ejecutar todos estos componentes simultáneamente se debe crear un fichero llamado docker-compose.yaml donde se especificarán los contenedores a ejecutar y así como otros parámetros.



```yaml
version: '3.1'
services:
  mongo:
    image: mongo
    restart: always
    ports:
      - 8094:27017
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: example
  mongo-express:
    image: mongo-express
    restart: always
    ports:
      - 8081:8081
    environment:
      ME_CONFIG_MONGODB_ADMINUSERNAME: root
      ME_CONFIG_MONGODB_ADMINPASSWORD: example
      ME_CONFIG_MONGODB_URL: mongodb://root:example@mongo:27017/
  broker:
    image: emqx/emqx
    restart: always
    environment:
    - "EMQX_NAME=emqx"
    - "EMQX_HOST=node1.emqx.io"
    ports:
      - "1883:1883"
      - "18083:18083"
```

Figura 6.3: Archivo docker-compose.yaml

Como se puede apreciar en la figura 6.3, los contenedores a ejecutar se encuentran dentro de la sección *services*. La definición de un contenedor empieza con la asignación de un nombre significativo, posteriormente se deben especificar algunos parámetros:

- *image:* representa el nombre con el que se ha empaquetado y distribuido una aplicación.

- *restart:* se refiere a la política de reinicio del contenedor para determinar cómo se debe comportar Docker en caso de que el contenedor falle o se detenga por algún motivo.
- *ports:* asigna un puerto del sistema donde está instalado el motor docker a un puerto del contenedor de la siguiente manera:

  <puerto_sistema>:<puerto_contenedor>.

  Por ejemplo, si queremos conectarnos a la base de datos desde fuera de la red creada por Docker Compose se debe hacer a través del puerto 8094. En cambio, cualquier contenedor que esté dentro de esa red lo hará a través del puerto 27017.
- *environment:* es una sección para especificar distintas variables de entorno, muchas veces necesarias para el funcionamiento de la aplicación empaquetada por el contenedor. En cierto modo son comparables a los parámetros que se especifican en la ejecución de un programa. Por ejemplo, en el caso de MongoDB es necesario especificar un usuario y una contraseña con los que conectarse a la base de datos, como Mongo Express necesita hacer precisamente eso, se le debe indicar cual es ese usuario y esa contraseña.

  Cabe recalcar que especificar este tipo variables directamente en los ficheros de configuración para el despliegue es una mala práctica, sobre todo si ese fichero se subirá a algún tipo de repositorio para la gestión de versiones. Para reducir los riesgos de seguridad, se puede crear un archivo .env y declarar esas variables para posteriormente referenciarlas desde el archivo docker-compose. Otra alternativa sería crear un secreto, si se utiliza como repositorio GitHub, el funcionamiento es similar al caso anterior.

6.4 Conexión del servicio de gestión con MongoDB

Una de las dependencias que se ha insertado en el «pom.xml» fue «*spring-boot-starter-data-mongodb*», esta librería incluye las funcionalidades básicas para trabajar con MongoDB así como una serie de configuraciones predefinidas para facilitar la conexión con la base de datos.

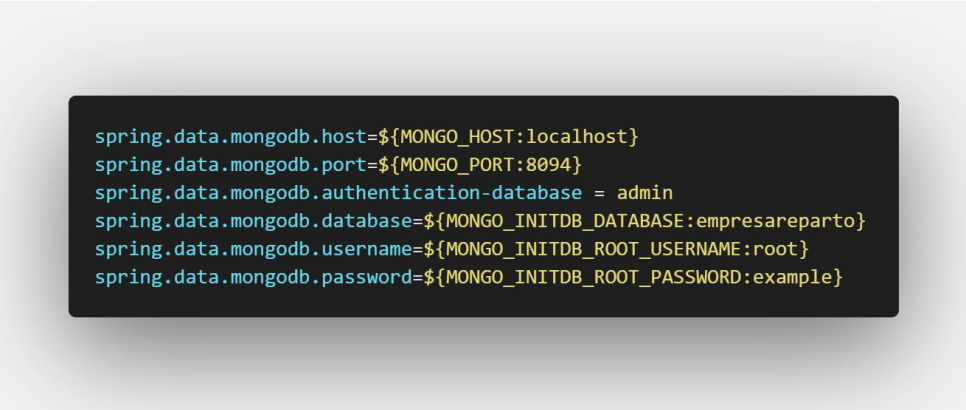


Figura 6.4: Propiedades para la conexión del servicio de gestión con MongoDB

El uso de la librería mencionada anteriormente reduce la configuración de la conexión a especificar séis propiedades dentro del archivo de configuración del proyecto «resources/application.properties», tal y como se puede apreciar en la figura 6.4. El texto de color azul hace referencia al nombre de la variables que Spring Boot necesita para crear la conexión. En cambio, el texto de color amarillo representa el valor de dichas variables pero, con un pequeño matiz. Muchos de los valores siguen una estructura del tipo:

${<variable-de-entorno>:<valor-por-defecto>}

Esto significa que Spring Boot primero intentará encontrar una variable de entorno con el nombre asignado para recoger su valor y en caso de que no la encuentre, asignará el por defecto. La conexión por defecto se hará con la máquina actual, localhost, a través del puerto 8094 que expone el puerto 27017 de la base de datos.

6.5 Conexión del servicio de gestión con EMQX

Para establecer una conexión MQTT con el bróker EMQX es necesario crear una instancia de la clase MQTTClient disponible en la librería «*org.eclipse.paho.client.mqttv3*». Al igual que con la base de datos, se requieren ciertas propiedades, concretamente la dirección y puerto del broker.

```
messaging.broker.host=${MQTT_BROKER:localhost}
messaging.broker.port=${MQTT_PORT:1883}
```

Figura 6.5: Propiedades para la conexión del servicio de gestión con EMQX

Las propiedades mostradas en la figura 6.5 también serán declaradas dentro del archivo «application.properties» Dado que esta librería tiene multitud de clases y métodos de los cuales solo se utilizará una pequeña parte y también teniendo en cuenta que uno de los objetivos de la arquitectura hexagonal es escribir que el núcleo de la aplicación sea independiente de tecnologías concretas, se ha aplicado un patrón fachada. La fachada consistirá de tres clases: *MessagingClient* que envuelve una instancia de *MQTTClient*, *Listener* que hereda de *MessagingClient* y cuya única funcionalidad es suscribirse y cancelar la suscripción y *Publisher* que solamente podrá publicar mensajes. Dentro de la capa de infraestructura se ha creado un directorio «config/messaging» que alberga la clase *MessagingClient* que actúa como base para la fachada y la clase *TopicsManager* que contiene los diferentes tópicos necesarios para el envío y recepción de mensajes MQTT.

```java
@Component
public class MessagingClient {
   private IMqttClient client;

   MessagingClient(@Value("${messaging.broker.host}") String
broker,
         @Value("${messaging.broker.port}") int port) throws
Exception {
      IMqttClient client;
      var options = new MqttConnectOptions();
      options.setCleanSession(true);
      options.setKeepAliveInterval(30);
      String host = "tcp://" + broker + ":" + port;
      client = new MqttClient(host, "ms-gestion");
      client.connect(options);
      this.client = client;
   }

   public IMqttClient getClient(){
      return client;
   }

}
```

Figura 6.6: Clase MessagingClient

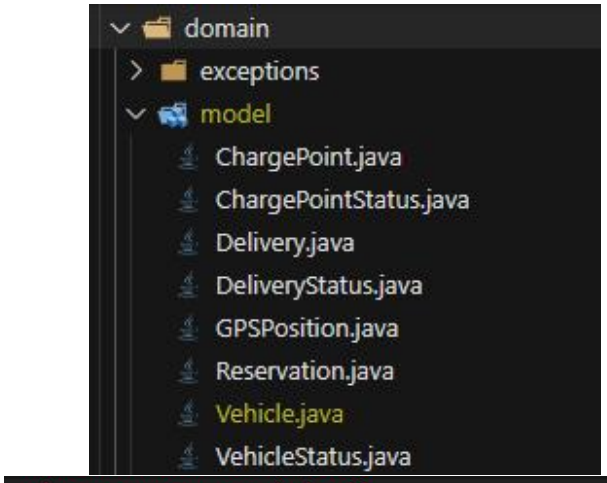
La anotación *@Component* que aparece en la figura 6.6 , se utiliza para indicarle al *framework* que genere una instancia de esta clase automáticamente al iniciar la aplicación y la use inyecte donde sea necesario. El servicio contendrá varias instancias de las clases *Listener* y *Publisher*, que a su vez guardan una instancia de *MessagingClient*, por tanto todas las operaciones que hagan esas clases son sobre la misma instancia de *MQTTClient*.

6.6 Desarrollo del servicio de gestión

Como se pudo ver al inicio de este capítulo, la arquitectura empleada consta de 3 capas: dominio, aplicación e infraestructura. Detallaremos el desarrollo del servicio de gestión capa por capa.

6.6.1. Capa de dominio

Todas las clases pertenecientes a esta capa son las detalladas en el capítulo anterior con la adición de algunas excepciones que se utilizarán en la capa de aplicación.




```java
package com.empresareparto.app.domain.model;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.Data;
import lombok.NoArgsConstructor;

@Document(collection = "Deliveries")
@Data @NoArgsConstructor
public class Delivery {
    @Id
    private String id;
    private GPSPosition pickupPosition;
    private GPSPosition deliverPosition;
    private DeliveryStatus status = DeliveryStatus.WAITING_PICKUP;
    private boolean assigned = false;

    public Delivery(GPSPosition pickupPosition, GPSPosition deliverPosition) {
        this.pickupPosition = pickupPosition;
        this.deliverPosition = deliverPosition;
    }

    public void complete() {
        setStatus(DeliveryStatus.DELIVERED);
    }
}
```

Figura 6.7: Capa de dominio

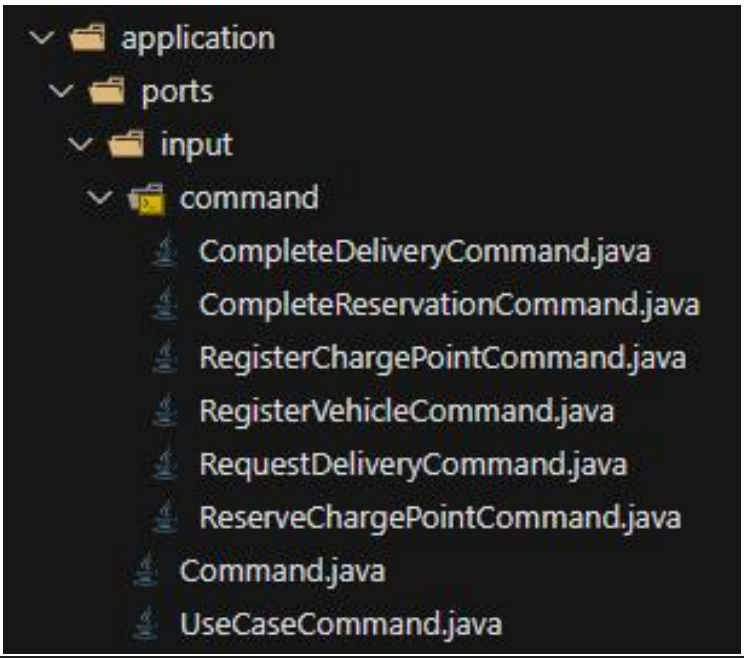
En la implementación de las clases de esta capa destaca el uso de las anotaciones siguientes:

- *@Document:* indica que la clase se almacena como un documento dentro de una colección en MongoDB.
- *@Id:* indica a la base de datos que el atributo marcado es el identificador del documento dentro de una colección.
- *@Data:* genera el código más recurrente de las clases, esto es, *getters*, *setters*, método *toString()* entre otros.
- *@NoArgsConstructor:* genera un constructor vacío.

6.6.2. Capa de aplicación

En esta capa se encuentran los puertos de entrada y salida, casos de uso y servicios.

Puertos Los puertos son los únicos puntos de entrada a la capa de aplicación y por tanto a los casos de uso. Tomando como base la implementación mostrada en [3] los puertos de entrada se han implementado siguiendo el patrón de diseño Comando, por tanto cada uno de estos puertos será una implementación de la interfaz *UseCaseCommand*, la cual debe tener un método *execute()* y extiende de *Command*.




```java
package com.empresareparto.app.application.ports.input.command;

import java.util.Optional;

import com.empresareparto.app.application.ports.input.UseCaseCommand;
import com.empresareparto.app.domain.model.Delivery;
import com.empresareparto.app.domain.model.GPSPosition;

import lombok.Data;

@Data
public class RequestDeliveryCommand implements UseCaseCommand<Delivery> {

    private GPSPosition pickup;
    private GPSPosition deliver;

    public RequestDeliveryCommand(GPSPosition pickup, GPSPosition deliver) {
        this.pickup = pickup;
        this.deliver = deliver;
    }

    @Override
    public Optional<Delivery> execute() {
        return Optional.of(new Delivery(pickup, deliver));
    }

}
```

Figura 6.8: Puertos de entrada en la capa de aplicación

Un comando debe ser autosuficiente, es decir, debe tener toda la información necesaria para ejecutar su acción. Una gran ventaja de la utilización de este patrón es que la capa de infraestructura no necesita conocer la implementación de los comandos, solamente la información que requieren. Otra ventaja es que facilita

la validación de datos de entrada, aunque en nuestro se ha preferido mantener estas clases con la menor complejidad posible y se ha dejado la validación fuera.

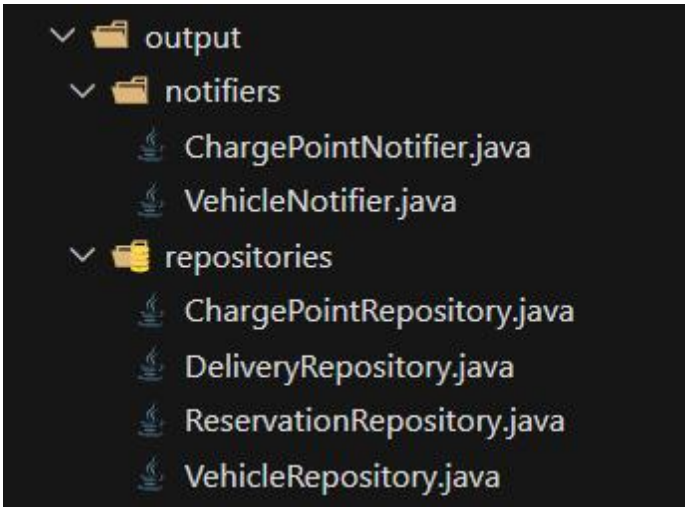


Figura 6.9: Puertos de salida en la capa de aplicación

VehicleRepository.java

```java
package com.empresareparto.app.application.ports.output.repositories;

import java.util.Collection;
import java.util.Optional;

import org.springframework.stereotype.Repository;

import com.empresareparto.app.domain.model.Vehicle;

@Repository
public interface VehicleRepository {
    public Optional<Vehicle> findById(String id);
    public Vehicle findByDeliveryId(String id);
    public Collection<Vehicle> findAll();
    public Vehicle save(Vehicle vehicle);
    public void deleteById(String id);
    public Collection<Vehicle> getAvailableVehicles();
}
```

VehicleNotifier.java

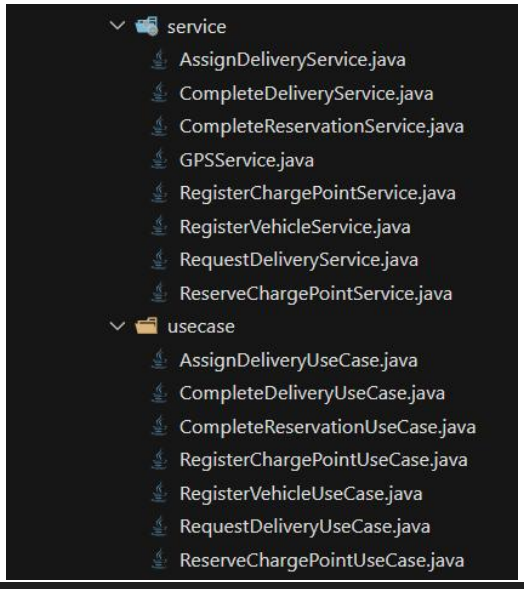
```java
package com.empresareparto.app.application.ports.output.notifiers;

import org.springframework.stereotype.Component;

import com.empresareparto.app.domain.exceptions.DeliveryNotificationException;
import com.empresareparto.app.domain.model.Delivery;
import com.empresareparto.app.domain.model.Vehicle;

@Component
public interface VehicleNotifier {
    public void notifyDelivery(Vehicle vehicle, Delivery delivery) throws DeliveryNotificationException;
}
```

Figura 6.10: Puertos de salida en la capa de aplicación

Entre los puertos de salida podemos encontrar dos clases de objetos, repositorios y notificadores. Los repositorios vienen del patrón de diseño con el mismo nombre, concebido para abstraer la lógica de acceso a la base de datos , por lo que encaja perfectamente con la arquitectura utilizada. En la figura se puede apreciar

el uso de la anotación *@Repository*, la cual es muy similar a *@Component*, pero orientada a clases con acceso a datos.

Los notificadores por su parte, se crearon con la intención de proporcionar una salida agnóstica tecnológicamente a los mensajes(repartos y reservas) dirigidos a vehículos y puntos de carga. El objetivo del uso de estas anotaciones es, de nuevo, que el *framework* inyecte una instancia de los repositorios y notificadores en las partes del código que se usen estas interfaces.




```java
@Service
@Transactional
public class AssignDeliveryService implements AssignDeliveryUseCase {

    private DeliveryRepository deliveryRepository;
    private VehicleRepository vehicleRepository;
    private VehicleNotifier notifier;

    @Override
    public void assignDelivery() throws Exception {

        List<Delivery> deliveries = (List<Delivery>) deliveryRepository.findUnassigned();
        List<Vehicle> vehicles = (List<Vehicle>) vehicleRepository.getAvailableVehicles();
        if (deliveries.isEmpty())
            throw new AssignDeliveryException(message:"No deliveries available");
        if (vehicles.isEmpty())
            throw new AssignDeliveryException(message:"No vehicles available");

        int maxAssignable = Math.max(deliveries.size(), vehicles.size());

        for (int i = 0; i < maxAssignable; i++) {
            Delivery delivery = deliveries.get(i);
            delivery.setAssigned(assigned:true);
            Vehicle v = vehicles.get(i);
            v.assignDelivery(delivery);
            notifier.notifyDelivery(v, delivery);
            vehicleRepository.save(v);
            deliveryRepository.save(delivery);
        }
    }

}
```

Figura 6.11: Casos de uso y servicios

Casos de uso Los casos de uso son aquellos requisitos definidos en la fase de análisis para cada uno de los componentes. A nivel de software se convierten en interfaces cuyos métodos reciben como parámetro un comando. Observando la

carpeta donde están situados se puede saber en poco tiempo de lo que es capaz el sistema. Los casos de uso definidos para el sistema de gestión son los mostrados en la parte inferior de la figura 6.11.

Servicios Los servicios son la implementación de los casos en su mayoría, aunque puede haber algún servicio como GPSService que puede actuar simplemente como una clase de utilidades para otros servicios. Cada servicio puede implementar uno o más casos de uso pero, es preferente seguir una relación uno a uno. La lógica de negocio se concentra en los servicios, en este punto se requieren interacciones con la base de datos, envío de mensajes u otro tipo de interacciones con agentes externos. Todas esas interacciones se llevan a cabo por medio de las interfaces definidas como puertos de salida(repositorios y notificadores).

En la implementación de los servicios se han empleado las anotaciones:

- *@Service:* esta anotación declara la clase como un servicio, haciendo que esté disponible para la inyección de dependencias en otras clases.
- *@Transactional:* esta anotación aplicada a una clase hace que todos los métodos públicos de esa clase sean gestionados como transacciones. Esto implica que Spring se encarga de iniciar, confirmar o revertir(en caso de excepción) una transacción.

6.6.3. Capa de infraestructura

En esta capa se pueden encontrar diferentes adaptadores, los cuales implementan los puertos tanto de entrada como de salida. Al ser implementaciones de los puertos, los adaptadores son inyectados en los servicios.

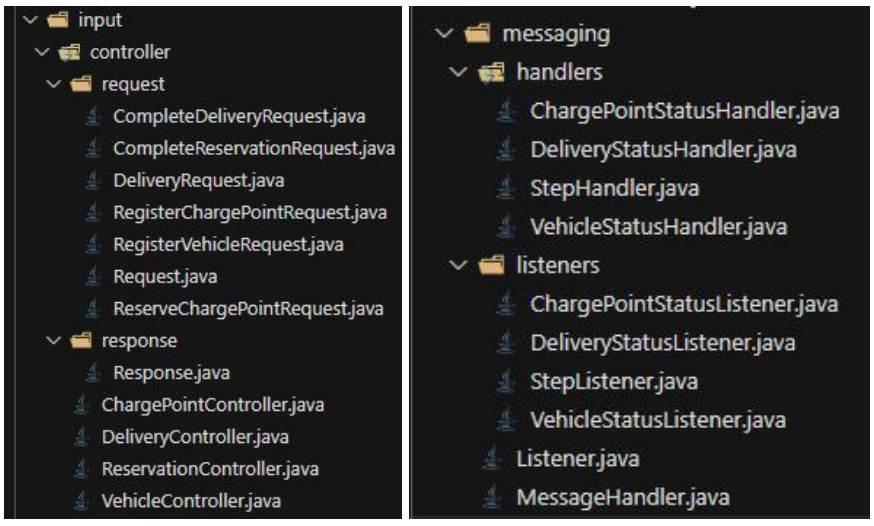


Figura 6.12: Adaptadores de entrada en la capa de persistencia

Adaptadores de entrada Se pueden distinguir dos tipos de adaptadores de entrada:

- *Controllers:* son componentes encargados de procesar las peticiones HTTP entrantes y devolver una respuesta. En el patrón arquitectónico REST, una petición tiene como objetivo la solicitud de un cierto servicio. Todo *controller* expone al exterior uno o varios *endpoints*(enlace en el que se solicita un servicio determinado). En nuestra implementación los controladores hacen referencia a las clases más importantes: *Vehicle,ChargePoint,Delivery y Reservation*. Dentro de cada uno de ellos intervienen tres tipos de objetos, excluyendo las excepciones. Estos son:

  - *Request*: Para evitar exponer los comandos de la capa de aplicación se ha creado esta interfaz que obliga a cualquier clase que la implemente a implementar un método genérico *toCommand()* que genere un comando a partir de los datos recibidos en el objeto *request*. Por tanto existirá una implementación específica de la interfaz para método que implique escritura.
  - *Service*: el controlador invocará el servicio solicitado de la capa de aplicación, proporcionándole como parámetro el objeto resultante de la invocación del método *toCommand* sobre el objeto *request*. Si el servicio no lanza ninguna excepción, el controlador devolverá un objeto de tipo *Response*.
  - *Response*: Es una clase que hereda de *ResponseEntity*, que solo tiene un método estático llamado *build* que al invocarlo contiene un código de respuesta HTTP y un objeto de tipo *Object*. La razón de utilizar la clase Object es que la clase Response se pueda utilizar tanto si la ejecución del servicio es exitosa y se quiera enviar el resultado, como si se una de las excepciones personalizadas del dominio y se quiera enviar el mensaje de esta.

```java
@RestController
@RequestMapping(path = "/reservation")
public class ReservationController {

    private ReserveChargePointService reserveChargePointService;
    private CompleteReservationService completeReservationService;

    @PostMapping
    public Response reserveChargePoint(@RequestBody ReserveChargePointRequest req) {
        try {
            return Response.build(reserveChargePointService.reserveChargePoint(req.toCommand()), HttpStatus.OK);
        } catch (ReserveChargePointException e) {
            return Response.build(e.getMessage(), HttpStatus.BAD_REQUEST);
        }
    }

    @PostMapping(path = "/complete")
    public Response complete(@RequestBody CompleteReservationRequest req) {
        try {
            return Response.build(
                    completeReservationService.execute(req.toCommand()),
                    HttpStatus.OK);

        } catch (Exception e) {
            return Response.build(e.getMessage(), HttpStatus.BAD_REQUEST);
        }
    }

}
```

Figura 6.13: Implementación del controlador para reservas

Dentro de la implementación de este tipo de adaptadores destacan las siguientes anotaciones:

- *@RestController*: se utiliza para crear un controlador que maneja las solicitudes HTTP y también especifica que los resultados devueltos por los métodos del controlador deben serializarse automáticamente en formato JSON u otro formato, y luego enviarse como respuesta HTTP al cliente.
- *@RequestBody*: las peticiones HTTP de tipo POST que aceptan nuestro controladores requieren que su cuerpo contenga un objeto de tipo JSON(*Javascript Object Notation*). Ese objeto JSON debe ser convertido en un objeto de Java para poder operar con él, normalmente esto se haría implementando un *mapper* para cada tipo de clase, pero gracias a esta anotación esta tarea se hace de forma automática ya que internamente utiliza la librería *Jackson*. Usar esta anotación implica tener un constructor vacío y los *setters* de todos los atributos de la clase a la que se desea convertir el cuerpo de la petición. En caso de que la conversión falle por la falta de lo anterior o por la falta de un atributo, se lanzará una excepción.
- *RequestMapping*: a nivel de clase, determina la ruta base de los métodos de esa clase. También se puede utilizar a nivel de método individual para especificar la ruta y con que tipo de petición HTTP se debe activar.
- *PostMapping*: es similar a RequestMapping, con la diferencia de que es específica a las peticiones de tipo Post.
- *@Log4j2*: proporciona una implementación básica de un *logger*.

- *Listeners:* se suscriben a diferentes tópicos proporcionados por la clase *TopicsManager*. Tienen además una instancia de la interfaz MessageHandler que extiende a *IMqttMessageListener*.

```java
@Component @Log4j2
public class ChargePointStatusHandler implements MessageHandler {

    private ChargePointRepository chargePointRepository;
    private ObjectMapper mapper;

    @Autowired
    public ChargePointStatusHandler(ChargePointRepository chargePointRepository, ObjectMapper mapper) {
        this.chargePointRepository = chargePointRepository;
        this.mapper = mapper;
    }

    @Override @Transactional
    public void messageArrived(String topic, MqttMessage message) throws Exception {
        try {
            ChargePoint cp = mapper.readValue(new String(message.getPayload()), ChargePoint.class);
        chargePointRepository.save(cp);
        } catch (Exception e) {
            log.error("STATE UPDATE FAILED: " + e.getMessage());
        }
    }

}
```

Figura 6.14: Implementación del manejador del esto de puntos de carga

```java
@Component @Log4j2
public class StepHandler implements MessageHandler {
    private AssignDeliveryService service;

    public StepHandler(AssignDeliveryService service) {
        this.service = service;
    }

    @Override
    public void messageArrived(String topic, MqttMessage message) throws Exception {
        try {
            service.assignDelivery();
        } catch (Exception e) {
            log.error(e.getMessage());
        }

    }

}
```

Figura 6.15: Implementación del manejador de pasos

Cada uno de estos manejadores determina el tratamiento que recibirán los mensajes que se reciban del tópico al que se suscribió el listener. Dado que los mensajes que tratan tres de los cuatro manejadores se deben almacenar en la base de datos, se ha preferido que estos operen directamente sobre los repositorios en lugar de crear un caso de uso y servicio para cada uno de ellos, ya que no llevaría asociada ninguna regla de negocio. El único manejador que invoca un servicio es el que recibe los pasos del simulador, con cada paso intenta asignar un reparto.

Adaptadores de salida Los adaptadores de salida implementados son los siguientes:

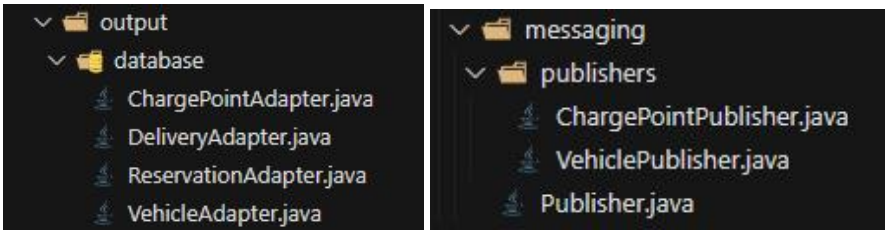


Figura 6.16: Adaptadores de salida

- Adaptadores de repositorios: normalmente deberían ser clases que implementan las interfaces repositorio de manera específica, aunque han acabado siendo también una interfaz, hay una razón detrás de ellos.

```java
public interface ReservationAdapter extends ReservationRepository, MongoRepository<Reservation, String> {

}
```

Figura 6.17: Implementación del repositorio de reservas

Gracias a la librería de Mongodb que se ha añadido al principio dentro del archivo «pom.xml» solo hace falta que estas interfaces extiendan los repositorios y a la interfaz MongoRepository, de esta forma los métodos de los repositorios serán implementados automáticamente. El requisito para

que esto funcione es que ambas sigan las mismas convenciones en cuanto a nombre de los métodos. En caso de necesitar un método que no siga las convenciones o de mayor complejidad, se puede utilizar la anotación @Query y escribir dentro la consulta requerida.

- *Publishers*: Se trata de varias clases que extienden la clase *Publisher* que forma parte de la fachada mencionada unas secciones atrás. Estas clases se encargan de serializar las reservas y repartos como texto plano y enviarlas a un tópico destino.

6.7 Desarrollo de los puntos de carga y vehículos

El desarrollo ha seguido la misma línea que con el servicio de gestión por lo que no se considera necesario detallar cómo se ha aplicado la arquitectura hexagonal en estos casos. Pero sí hay un detalle a nivel de implementación que no se ha visto anteriormente, la realización de peticiones REST a otros servicios. Los vehículos y puntos de carga realizaban este tipo de comunicación en varias ocasiones: el registro, solicitar reserva, completar reserva y completar reparto. Dentro de la capa de infraestructura de ambos proyectos se ha implementado una clase *RestClient*.

```java
@Component
@Log4j2
public class RestClient {

    private WebClient client;
    private String registerCpURI = "upvexpress/chargepoints";
    private String completeReservationURI = "upvexpress/reservation/complete";

    @Autowired
    public RestClient(@Value("${msgestion.host}") String host, @Value("${msgestion.port}") String port) {
        client = WebClient.builder().baseUrl("http://" + host + ":" + port).build();
    }

    public void register(String id, GPSPosition pos) {
        RegisterCPRequest request = new RegisterCPRequest(id, pos);
        ChargePoint returnedCP;
        ResponseSpec resp = client.post().uri(registerCpURI).bodyValue(request).accept(MediaType.APPLICATION_JSON)
                .retrieve();
        try {
            log.info("creado request: " + id);
            returnedCP = resp.bodyToMono(elementClass:ChargePoint.class).block();
            log.info("returned chargepoint: " + returnedCP.toString());
        } catch (Exception e) {
            log.info(resp.bodyToMono(elementClass:String.class).block().toString());
        }

    }
```

Figura 6.18: Clase RestClient en un vehículo

En esta clase se usan dos anotaciones nuevas:

- *@Autowired:* se utiliza para inyectar una dependencia de un bean Spring en una propiedad o método. Se puede aplicar a nivel de atributo o de constructor.

- *@Value:* se utiliza para inyectar un valor literal en una propiedad o método. En este caso esos valores son extraídos del archivo «application.properties».

En la petición REST han intervenido las siguientes clases:

- *WebClient:* es una clase para realizar solicitudes HTTP a servicios web. La clase proporciona una API sencilla para realizar solicitudes GET, POST, PUT y DELETE. También proporciona una API para configurar las solicitudes HTTP, como las cabeceras, el contenido y el timeout.
- *ResponseSpec:* se utiliza para validar la respuesta de una solicitud HTTP. La clase proporciona una API para validar el código de estado, las cabeceras, el contenido y el cuerpo de la respuesta.
- *Mono:* representa un flujo de datos de un solo elemento. La clase se utiliza para representar datos que se obtienen de una fuente, como una solicitud HTTP..

6.8 Pruebas

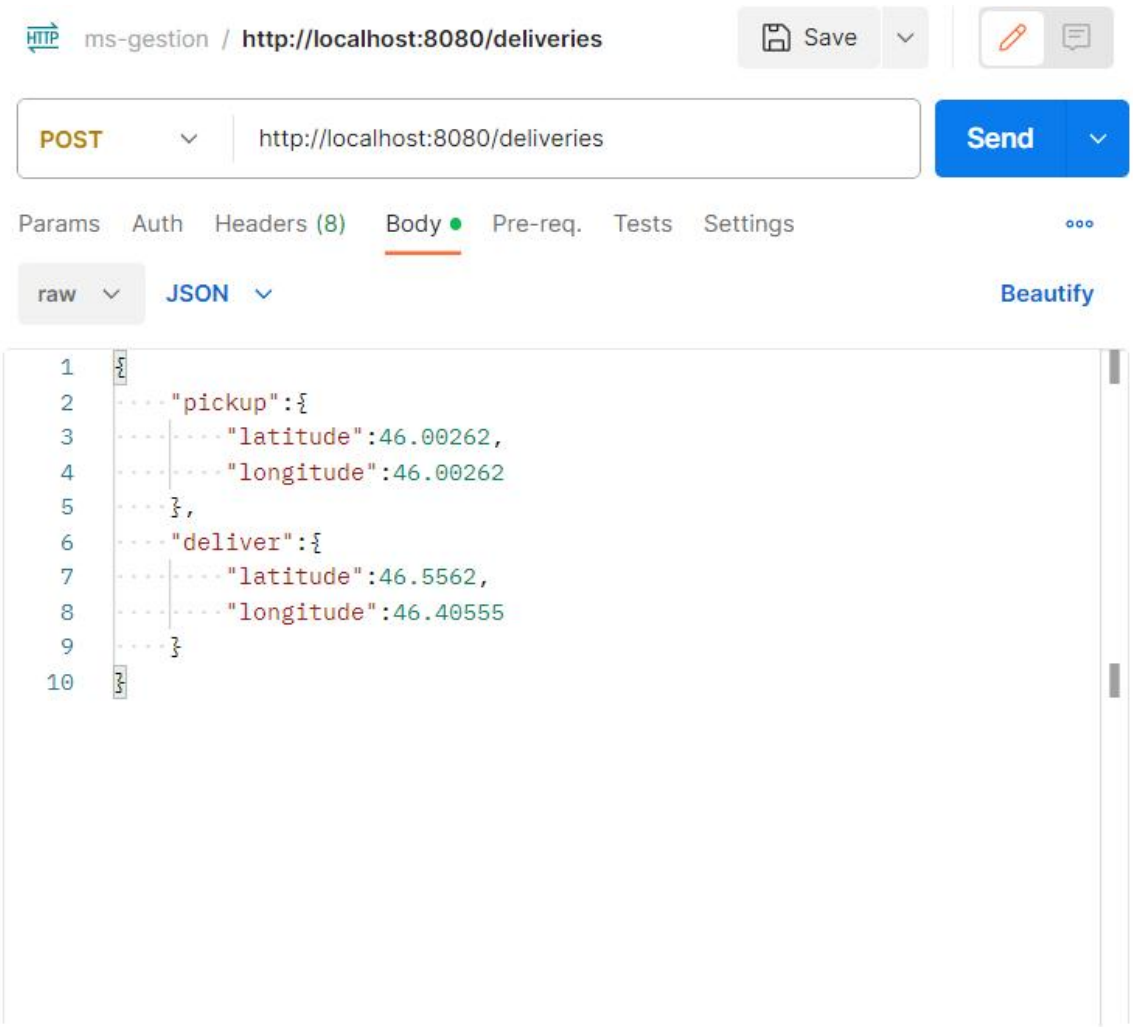


Figura 6.19: Petición para la solicitud de un reparto en PostMan

Al acabar de desarrollar un caso de uso, se comprobaba su correcto funcionamiento. Las pruebas se hacían desplegando los componentes como la base de

datos y bróker utilizando un archivo docker-compose y los componentes desarrollados se ejecutaban en el modo depuración de Visual Studio Code.

En el caso de las pruebas de los controladores se crearon una serie de peticiones Postman como la de la figura 6.19.

Si la respuesta no era la que se esperaba, se volvía al editor de código para establecer unos puntos de interrupción(*breakpoints*) en algunas líneas del método a probar para observar como avanza el valor de las variables y el flujo de ejecución.

En el caso de las comunicaciones por MQTT, el procedimiento era similar, pero con la facilidad de que EMQX permite la visualización de todos los clientes conectados y sus suscripciones. También se ha utilizado un *logger* para mostrar por terminal el estado de algunos objetos con cada paso del simulador.

6.9 Conexión de ThingsBoard con ThingsBoard Gateway

```
thingsboard:
  image: thingsboard/tb-postgres
  restart: always
  ports:
    - "9094:9090"
    - "9095:1883"
    - "7070:7070"
    - "5683-5688:5683-5688/udp"
  environment:
    TB_QUEUE_TYPE: in-memory
  volumes:
    - ./iot/tb-data:/data
    - ./iot/tb-logs:/var/log/thingsboard

tb-gateway:
  image: thingsboard/tb-gateway
  environment:
    - host=thingsboard
    - port=1883
    - accessToken=${TB_ACCESS_TOKEN}
  volumes:
    - "./iot/gw-config:/thingsboard_gateway/config"
    - "./iot/gw-extensions:/thingsboard_gateway/extensions"
    - "./iot/gw-logs:/thingsboard_gateway/logs"
  container_name: tb-gateway
  restart: always
```

Figura 6.20: Archivo docker compose para ejecución y conexión del gateway

Para establecer esta conexión es necesario iniciar ThingsBoard e iniciar sesión como administrador, con unas credencias por defecto. A continuación se debe ir a la sección de dispositivos y agregar un nuevo dispositivo, debemos marcar una casilla que pregunta si el dispositivo es un gateway. Una vez añadido el dispositivo se copia su token de acceso. Hecho esto, se debe ejecutar ThingsBoard gateway, aunque si se utilizan ambas aplicaciones dentro de contenedores docker resulta más sencillo crear un archivo docker-compose para ejecutarlos a la vez. Teniendo el token de acceso debemos entrar en los archivos de instalación de ThingsBoard Gateway y buscar dentro de la carpeta «config», un archivo llama-

do «tb-gateway.yaml». Dentro de ese archivo se deben especificar los parámetros host y port dentro de la sección *thingsboard* y el token de acceso en la subsección *security*. Si se utiliza docker compose la conexión se realizaría especificando los parámetros anteriores como variables de entorno tal y como se aprecia en la figura 6.20.

En caso de utilizar este enfoque debemos especificar unos volúmenes si se desea persistir sus configuraciones y datos en caso de destruir los contenedores.

6.10 Conexión de ThingsBoard Gateway con EMQX

La documentación oficial de ThingsBoard recomienda situar el gateway en la misma red, por tanto se ha fusionado el archivo anterior con el que ejecutaba MongoDB y EMQX. Al ejecutar la orden "docker compose up -d" todos los contenedores iniciarán su ejecución. Dentro de la carpeta «iot/gw-config» se deben haber creado varios archivos con extensión ".json", cada uno de ellos sirve para establecer una conexión mediante un protocolo específico. El archivo a buscar es «mqtt.json».

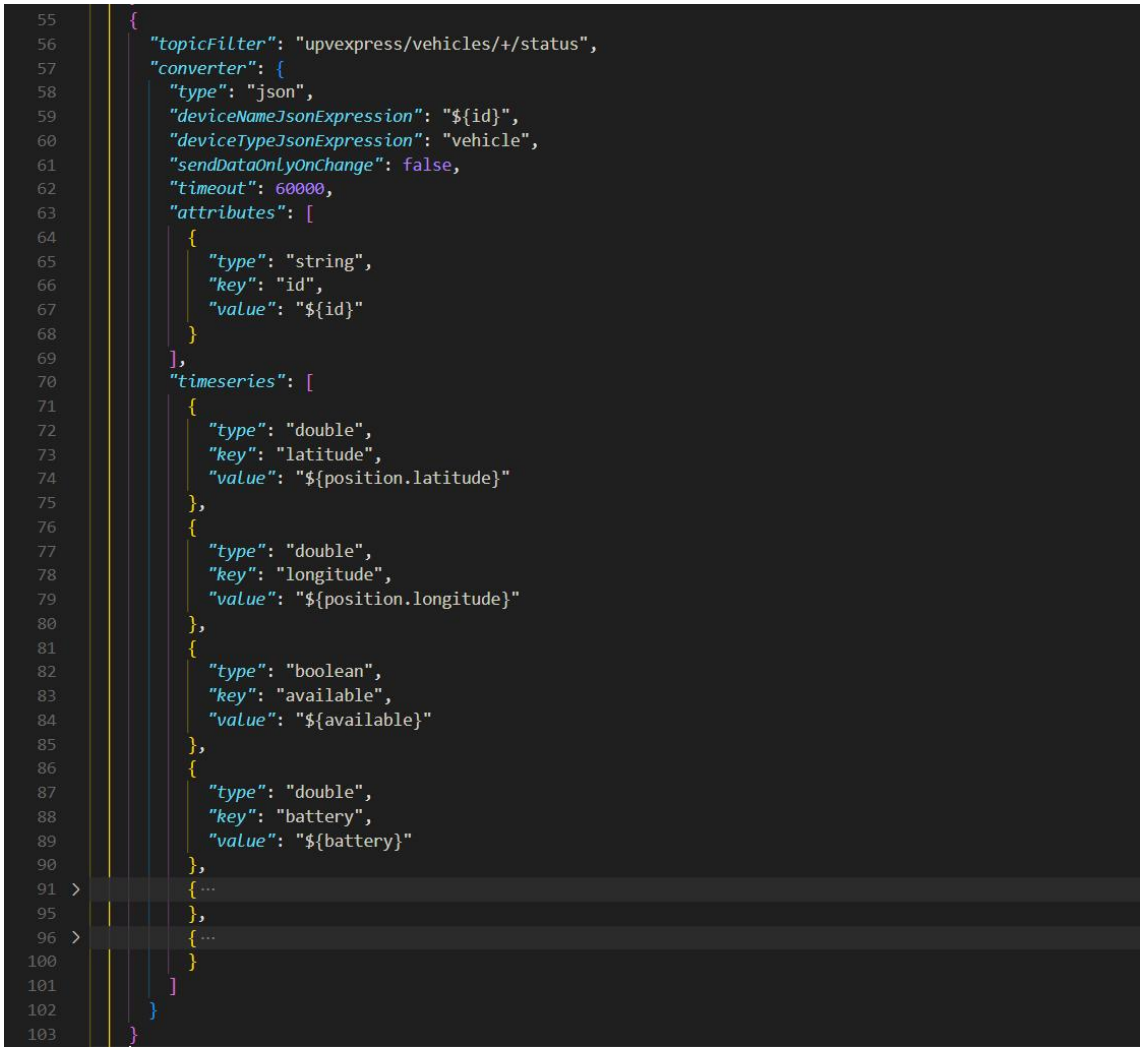
Figura 6.21: Archivo mqtt.json

Dentro de la sección *broker* se deben indicar los atributos host y port. En la figura 6.21 se puede ver que sus valores son broker y 1883. Con estos cambios aplicados lo único restante es reiniciar el contenedor con la orden "docker compose restart tb-gateway".

6.10.1. Extracción de datos de los tópicos MQTT

Con el gateway ya conectado al bróker, se puede proceder con la extracción de datos de los tópicos. Según se indica en la documentación oficial, el archivo

anterior presenta una sección llamada «mapping», la cual es una lista que contiene unos objetos JSON predefinidos que sirven de ejemplo a nuestra finalidad. Esos objetos adaptados a nuestro objetivo presentan la estructura presente en la figura que sigue:



```
{
  "topicFilter": "upvexpress/vehicles/+/status",
  "converter": {
    "type": "json",
    "deviceNameJsonExpression": "${id}",
    "deviceTypeJsonExpression": "vehicle",
    "sendDataOnlyOnChange": false,
    "timeout": 60000,
    "attributes": [
      {
        "type": "string",
        "key": "id",
        "value": "${id}"
      }
    ],
    "timeseries": [
      {
        "type": "double",
        "key": "latitude",
        "value": "${position.latitude}"
      },
      {
        "type": "double",
        "key": "longitude",
        "value": "${position.longitude}"
      },
      {
        "type": "boolean",
        "key": "available",
        "value": "${available}"
      },
      {
        "type": "double",
        "key": "battery",
        "value": "${battery}"
      },
      {…
      },
      {…
      }
    ]
  }
}
```


Figura 6.22: Objeto JSON para la extracción de datos

Cada uno de los atributos de este objeto tiene un propósito:

- *topicFilter:* representa el tópico MQTT al que debe suscribirse el gateway.
- *type:* hace referencia al formato de los mensajes se envían al tópico anterior.
- *deviceNameJsonExpression:* es el atributo del objeto JSON del mensaje que representa el nombre del dispositivo.
- *deviceTypeJsonExpression:* es el perfil de dispositivo que se debe aplicar en ThingsBoard.
- *attributes:* dentro del objeto JSON transmitido en el mensaje, referencia aquellos atributos que son inmutables con el paso del tiempo

- *timeseries:* dentro del objeto JSON transmitido en el mensaje, referencia aquellos atributos que cambian con el paso del tiempo, como la posición o porcentaje de batería.

Tanto *attributes* como *timeseries* contienen objetos JSON con los atributos a extraer con la estructura siguiente:

- *type:* representa el tipo de dato.
- *key:* la clave del atributo que se envía a ThingsBoard.
- *value:* la clave cuyo valor se envía a ThingsBoard.

6.11 Creación de paneles para la visualización de datos

Para crear un panel en ThingsBoard es necesario iniciar sesión como administrador propietario con las credenciales por defecto, navegamos a la sección Paneles, pulsamos en agregar panel, introducimos los datos requeridos y entramos en modo edición del panel. No se entrará en detalles sobre como añadir un widget o los tipos de widget que hay, ya que en la documentación oficial[38] se ofrece una descripción completa. Sin embargo, sí se profundizará en cómo se pueden visualizar los datos extraídos por ThingsBoard Gateway.

El primer paso es entrar al panel y crear un alias para el perfil del dispositivo(atributo deviceTypeJsonExpression). Cabe aclarar que si bien la plataforma permite la creación de perfiles, no es necesario hacerlo ya que al haber especificado estos en el archivo de configuración, se crean automáticamente. Lo mismo sucede con los propios dispositivos.

Figura 6.23: Creación de alias en ThingsBoard

La ventana que se muestra para crear un alias es la que aparece en la figura anterior. En el campo Tipo de dispositivo, debemos seleccionar el perfil de dispositivo del que se quiera extraer los datos. Una vez creado el alias se añade un widget, entramos en su modo de edición y agregamos una fuente de datos si no hay una. Al abrir el modo siguiente se muestra una ventana como la de la figura:

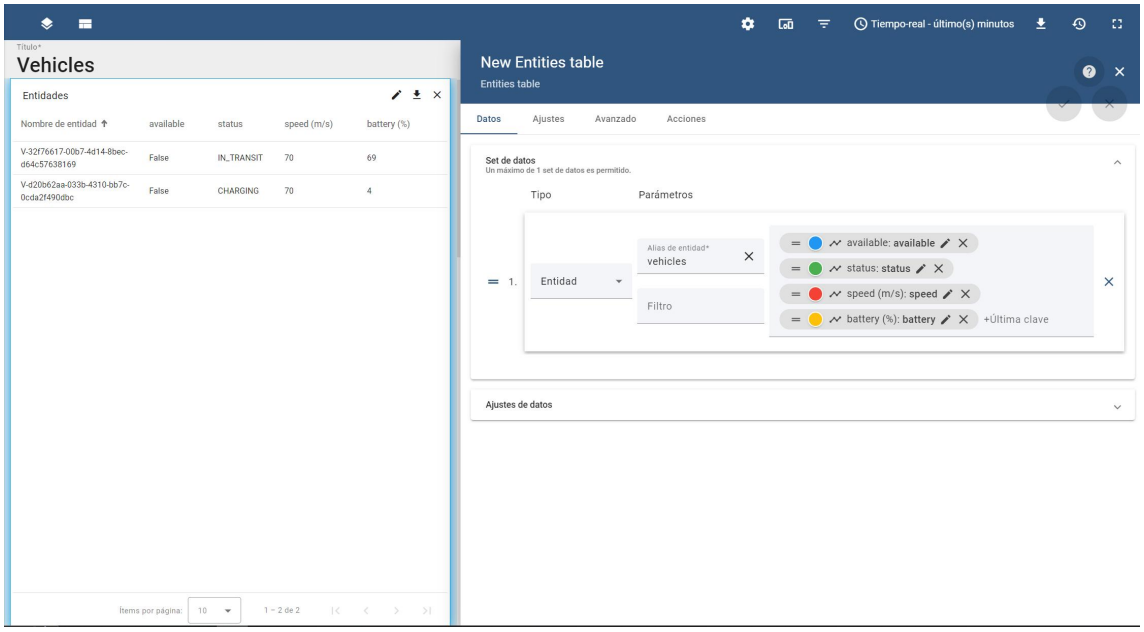


Figura 6.24: Configuración para la visualización de datos en un widget

Debemos especificar el alias creado anteriormente y seleccionar las claves de los atributos que habíamos especificado en las listas *attributes* o *timeseries* del archivo anterior. Al pulsar en ese campo aparece una lista desplegable con todas las claves, no es necesario memorizarlas. Este último paso se repetirá para todos los widgets que se deseen añadir al panel. Por limitaciones de tiempo, los paneles creados son una versión más básica de lo planeado inicialmente. Los resultados son los siguientes:

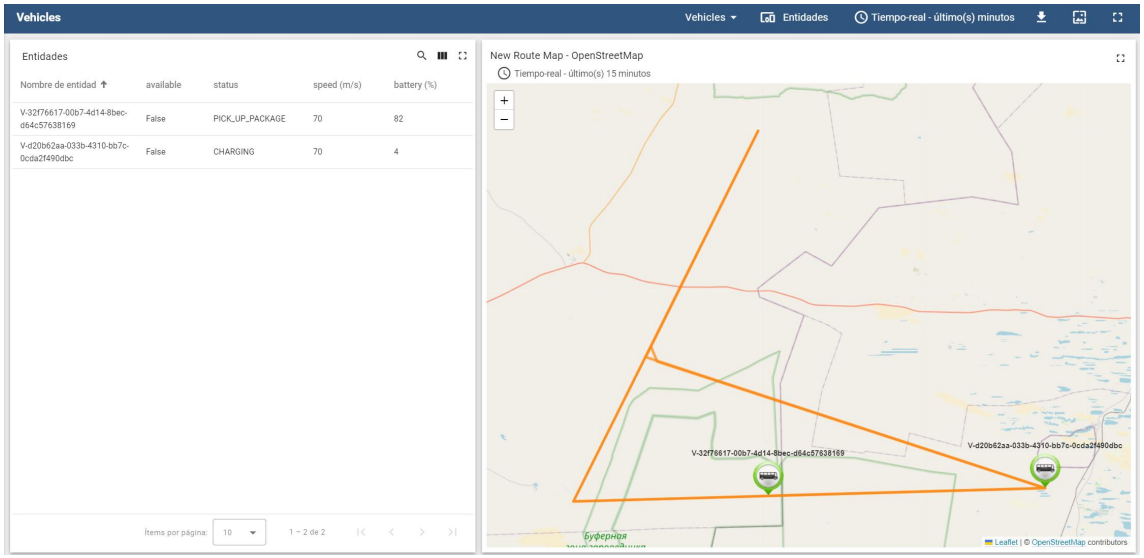


Figura 6.25: Panel para la visualización de vehículos

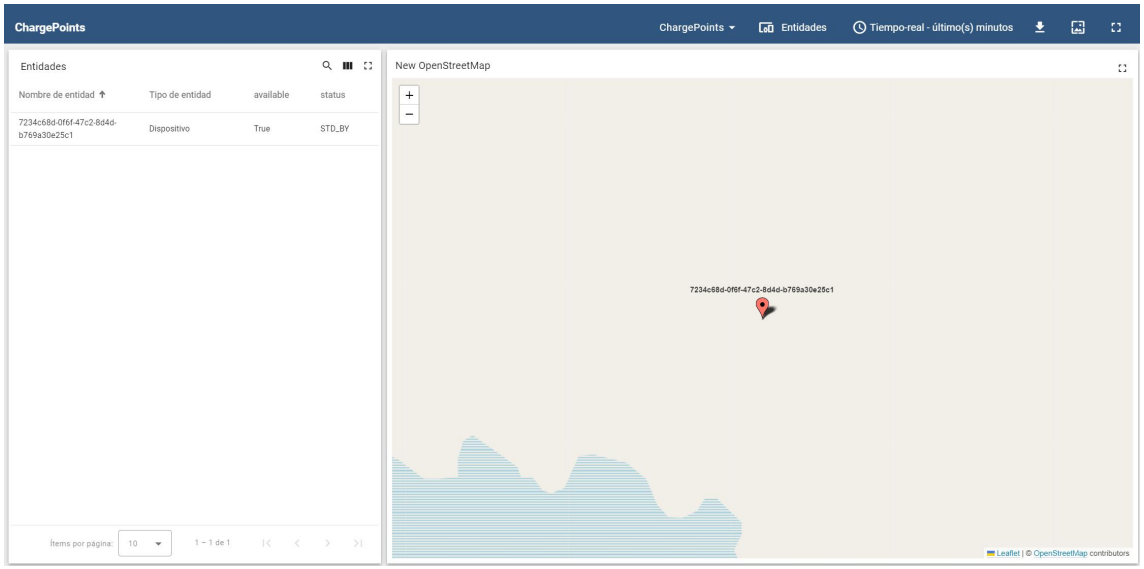


Figura 6.26: Panel para la visualización de puntos de carga

Vista la forma en la que se pueden extraer datos de un tópico MQTT a través del gateway, se podría elaborar también un panel para la visualización de los repartos a pesar de que estos no son un dispositivo. Sería necesario añadir la configuración para extraer los datos del tópico donde se envían los mensajes y crear un panel de la forma descrita.

CAPÍTULO 7
Implantación

7.1 Empaquetado de los componentes

Para empaquetar los proyectos desarrollados de Spring Boot en imágenes es necesario crear un archivo llamado «dockerfile» dentro de cada uno de los proyectos. Este tipo de archivos se escriben siguiendo la sintaxis especificada por Docker. Una pequeña parte de la sintaxis se puede apreciar en la figura siguiente:

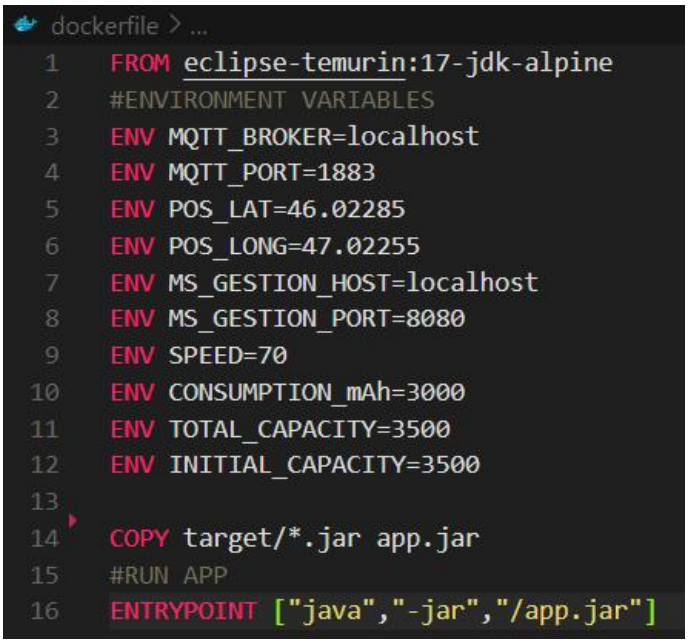


Figura 7.1: Dockerfile de un vehículo

En el dockerfile podemos encontrar los comandos siguientes:

- *FROM:* especifica la imagen del contenedor sobre la que se construirá la imagen actual. La imagen *ecplipse-temurin:17-jdk-alpine* proporciona una versión ligera de un JDK(Java Development Kit) instalada sobre alpine, una distribución ligera de Linux.

```
server.port=8085
speed.mps=${SPEED:70}
power_consumption=${CONSUMPTION_mAh:3000}
battery_capacity.total=${TOTAL_CAPACITY:3500}
battery_capacity.initial=${INITIAL_CAPACITY:3500}
position.lat=${POS_LAT:46.02285}
position.long=${POS_LONG:47.02255}
messaging.broker.host=${MQTT_BROKER:localhost}
messaging.broker.port=${MQTT_PORT:1883}
msgestion.host=${MS_GESTION_HOST:localhost}
msgestion.port=${MS_GESTION_PORT:8080}
```

Figura 7.2: application.properties de un vehículo

- *ENV:* esta instrucción crea una variable de entorno en la imagen. Todas las variables creadas son las mismas que se han definido como valor en el archivo application.properties .
- *COPY:* como su nombre indica, esta instrucción simplemente copia un archivo o directorio dentro de la imagen del contenedor. En este caso copia el archivo jar generado por como resultado de ejecutar el comando «./mvnw clean install»
- *ENTRYPOINT:* sirve para ejecutar un binario especificado con algunos argumentos predefinidos que proporciona la posibilidad de añadir otros al ejecutar el contenedor.

Al acabar de crear todos los archivos dockerfile se debe construir la imagen, para ellos abriremos un terminal en el directorio donde están localizados y ejecutar la orden:

docker build -t <nombre-imagen:etiqueta> .

La opción -t permite asignar un nombre a la imagen y opcionalmente una etiqueta y el punto hace referencia al directorio actual. Opcionalmente se pueden subir las imágenes al repositorio en línea de Docker, Dockerhub. Para ello se debe iniciar sesión y crear un repositorio, el nombre del repositorio tiene la siguiente forma: <nombre-de-usuario>/<nombre-repositorio>. Posteriormente se debe iniciar sesión en docker desde el terminal mediante el comando ”docker login” y renombrar la imagen de la siguiente forma:

```
docker tag nombre-imagen:etiqueta nombre-usuario/repositorio:etiqueta
```

Figura 7.3: Orden docker tag

Con esto se renombra la imagen y se puede subir al repositorio utilizando el comando:

```
docker push nombre-usuario/repositorio:etiqueta
```

Figura 7.4: Orden docker push

Si el repositorio que se ha creado es público, cualquier persona puede descargar y utilizar la imagen.

7.2 Persistencia de la configuración de ThingsBoard

ThingsBoard Gateway y ThingsBoard se pueden configurar modificando ciertos archivos, para automatizar al máximo el despliegue es deseable persistir sus configuraciones. En la actualidad ha surgido una nueva metodología basada en DevOps, llamada GitOps. Esta metodología consiste en usar los repositorios de Git para la gestión de infraestructura como código (IaC) y sus respectivas configuraciones. Estas prácticas se integran especialmente bien con plataformas como Kubernetes. Cualquier cambio en el código del repositorio inicia una serie de procesos automatizados que prueban y despliegan la última versión del código en un entorno de producción. La implementación de estas prácticas resulta interesante, pero debido a limitaciones de tiempo solamente se va a intentar un despliegue local de la infraestructura necesaria.

7.3 Despliegue con Minikube

Para hacer un despliegue con Kubernetes es conveniente saber que esta plataforma gestiona el estado de un clúster a través de objetos, a efectos prácticos, archivos de configuración. Estos objetos representan el estado deseado del clúster y pueden ser de los siguientes tipos:

- *Pods:* es la mínima unidad lógica gestionada por Kubernetes, representa una colección de uno o más contenedores. No es común escribir este tipo de archivos, se suelen utilizar los despliegues.
- Despliegues: se encarga de describir *pods* y *ReplicaSets*.
- *ReplicaSets:* este objeto se encarga de escalar el número de pods de un despliegue, esto se puede de hacer forma manual o automática.
- Servicios: son una manera de exponer los pods de un despliegue a la red, para que los clientes u otros *pods* puedan interactuar con ellos. Existen diferentes tipos de servicios:
    - *ClusterIP:* expone el servicio asignando una dirección IP solamente visible en la red interna del clúster, no es accesible desde el exterior. En caso de no especificar el tipo de un servicio, este es el que se asigna por defecto.

- *NodePort:* expone el servicio en un puerto fijo de los nodos del clúster, esto permite conexiones externas sin un utilizar un equilibrador de carga.
- *LoadBalancer:* expone el servicio al exterior del clúster utilizando un equilibrador de carga, se debe proveer uno o integrar Kubernetes con un proveedor de servicios en la nube.
- *ExternalName:* no se crea un servicio dentro del clúter sino que proporciona un nombre DNS para un servicio externo.

Para diseñar los despliegues y servicios se puede partir del archivo docker-compose creado anteriormente, para conocer los puertos que se deben exponer y la configuración de cada contenedor. También hay que definir el tipo de servicio que se debe a cada uno de los contenedores. Para no añadir complejidad se utilizará el tipo NodePort para aquellos servicios que ofrezcan alguna funcionalidad que debe ser accesible desde el exterior y ClusterIP para los que ofrezcan una funcionalidad a otros pods.

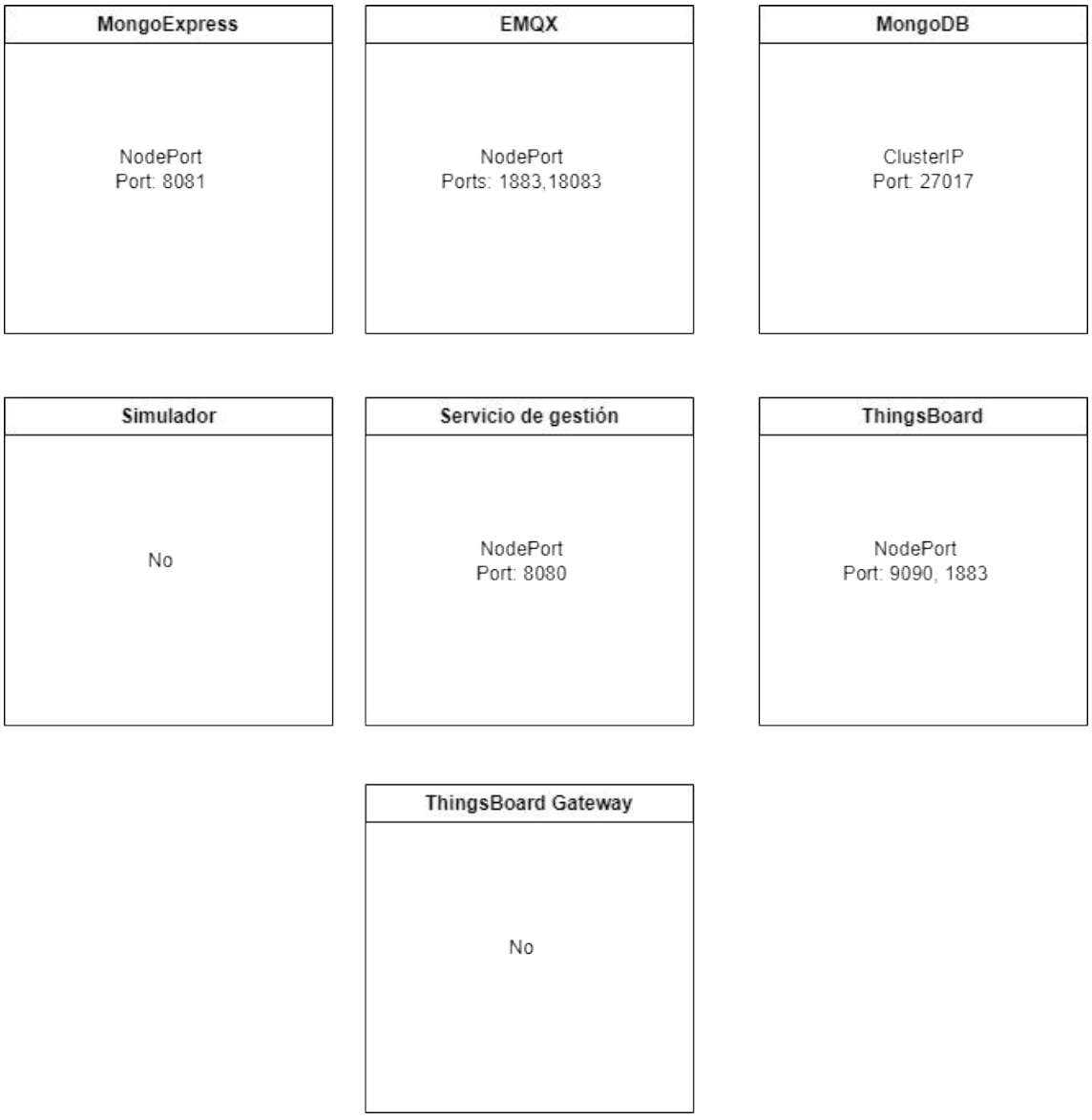


Figura 7.5: Tipos de servicio de los contenedores

Una vez clasificados los servicios solo resta escribir los archivos de despliegue y servicio. En el caso del servicio de gestión los archivo serían de despliegue y servicio serían los siguientes:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
  creationTimestamp: null
  labels:
    app: ms-gestion
  name: ms-gestion
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ms-gestion
  strategy: {}
  template:
    metadata:
      annotations:
      creationTimestamp: null
      labels:
        app: ms-gestion
    spec:
      containers:
        - env:
            - name: MONGO_HOST
              value: mongo
            - name: MONGO_PORT
              value: "27017"
            - name: MQTT_BROKER
              value: broker
            - name: MQTT_PORT
              value: "1883"
          image: catalinm00/tfg:ms-gestion.1.0.0
          name: ms-gestion
          ports:
            - containerPort: 8080
          resources: {}
      restartPolicy: Always
status: {}
```

Figura 7.6: Archivo de despliegue del servicio de gestión

A continuación se presenta el significado de los campos más importantes del archivo:

- *apiVersion*: este campo especifica la versión de la API de Kubernetes que se está utilizando. En este caso, se está utilizando la API apps/v1.
- *kind*: este campo especifica el tipo de objeto que se está creando. En este caso, se está creando un despliegue.
- *metadata*: contiene información sobre el despliegue. En este caso, se especifica el nombre del despliegue, ms-gestion, y la etiqueta app=ms-gestion que se utilizará para identificar los pods creados por el despliegue.

- *replicas*: especifica el número de pods que debe crear el despliegue. En este caso, se creará 1 pod.
- *selector*: se utiliza para identificar los pods creados por el despliegue. En este caso, se utilizará la etiqueta app=ms-gestion.
- *template*: este campo especifica la configuración para los pods creados por el despliegue.
- *spec*: este campo especifica la configuración del despliegue.
- *containers*: este campo especifica el contenedor que debe ejecutarse en cada pod.
- *env*: define las variables de entorno que deben establecerse para el contenedor.
- *ports*: especifica los puertos que debe exponer el contenedor.

Muchos de estos campos se repiten en los distintos tipos de archivos que se pueden escribir.

```yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: ms-gestion
  name: ms-gestion
spec:
  ports:
    - name: "8080"
      port: 8080
      targetPort: 8080
  selector:
    app: ms-gestion
  type: NodePort
status:
  loadBalancer: {}
```

Figura 7.7: Archivo servicio del despliegue

En este archivo destaca el campo ports que se estructura de la siguiente manera:

- *name:* es opcional y se utiliza para identificar el puerto.
- *port:* especifica el puerto del servicio.
- *targetPort:* especifica el puerto del contenedor al que se redirige el tráfico entrante desde el puerto de servicio.

Desafortunadamente, debido a problemas técnicos no se ha podido probar el funcionamiento del despliegue y exposición de los servicios.

7.4 Despliegue con Docker Swarm

A diferencia de minikube, este orquestador utiliza los archivos docker-compose para realizar los despliegues. Se han añadido los contenedores de vehículos y puntos de carga al archivo. Para realizar un despliegue es necesario crear el Swarm(clúster) y añadir uno o varios nodos. Los nodos pueden ser esclavos o maestros dependiendo de como se unan al clúster. Para iniciar un clúster se ejecuta la orden:

docker swarm init

La respuesta que se recibe son dos comandos:

- docker swarm join –token <token>: para unirse al swarm como esclavo
- docker swarm join-token manager: para unirse como maestro.

Por defecto, la instacia de docker que ejecute el comando init se convertirá en un nodo maestro. Habiendo creado el clúster solo resta ejecutar el comando:

docker stack deploy –compose-file docker-compose-yaml <nombre-stack>

El comando se ejecuta correctamente y se empiezan a ejecutar los contenedores, ahora surge la problemática de que cada contenedor tiene un tiempo de inicio diferente, por tanto si por ejemplo la base de datos no ha iniciado, el contenedor del servicio de gestión fallará y consecuentemente también todos los vehículos y puntos de carga. Con lo cual se necesita emplear mecanismos para comprobar el estado de salud de los contenedores(healthcheck). Docker Swarm proporciona un comando test, por desgracia no se ha dispuesto de tiempo suficiente para indagar en su funcionamiento. Cabe mencionar que Kubernetes también ofrece mecanismos similares e incluso más complejos.

CAPÍTULO 8
Conclusiones

Este trabajo ha supuesto unos retos muy interesantes tanto a nivel de diseño como a nivel de desarrollo. También ha aportado mucho a nivel tecnológico ya que se han reforzado conocimientos previos de Spring Boot y también se han adquirido otros sobre el uso e implementación automática de los repositorios, la conexión con una base de datos así como realizar peticiones REST a otro servicio. Se ha aprendido sobre el funcionamiento de MQTT y como se realiza la conexión con el bróker en un proyecto de Spring Boot. Además se ha tenido la oportunidad de aprender sobre el funcionamiento interno de Kubernetes y cómo se pude desplegar una solución utilizándolo. En cuanto a los objetivos marcados en un principio, estos eran:

- Gestionar las reservas de puntos de carga de manera automática.
- Optimizar la asignación de repartos a los vehículos disponibles para maximizar el tiempo operativo de la flota.
- Desarrollar unos microservicios que simulen el comportamiento de los componentes mencionados anteriormente.
- Establecer las diferentes comunicaciones entre los microservicios.
- Integrar una plataforma IoT para la monitorización y análisis de los datos que vehículos y puntos de carga van generando en tiempo real.
- Poner en producción el sistema utilizando un orquestador de contendores de forma local e idealmente en un proveedor de servicios en la nube.

Se han conseguido los objetivos de elaborar un sistema que gestione las reservas de puntos de carga y asignación de repartos de forma automática, también se ha conseguido implementar con éxito los microservicios que simulan el comportamiento de vehículos. Observando estos microservicios se ha echado en falta que estos tuvieran un comportamiento más dinámico y menos previsible, sobre todo en los vehículos, un factor que puede influir mucho en este aspecto es la velocidad la cual era constante. Aumentos significativos en la velocidad habría abierto las puertas a la implementación de alarmas en ThingsBoard. También el hecho de que un vehículo se quede sin batería puede ser un buen caso para una alarma.

A lo largo del desarrollo de este trabajo no se han encontrado muchas dificultades propiamente dichas, a excepción de una fase, pero sí se ha requerido de una cantidad considerable de tiempo en los siguientes aspectos:

- Aprendizaje y descubrimiento de las tecnologías: en este apartado destaca la búsqueda de plataformas IoT y la comparación entre ellas, seguido por el aprendizaje de como integrar estas plataformas con otros sistemas. También el aprendizaje de Kubernetes(compaginado con otras tareas) ha requerido alrededor de una semana completa para comprender su estructura y funcionamiento. Afortunadamente ya se disponía de una buena base en otras tecnologías como Spring Boot, Docker o Postman.
- Diseño de las comunicaciones: más que por complejidad, fue por el número de opciones disponibles y por el propio número de casos a diseñar, ya que había que nos servían tanto un modelo publicador suscriptor como petición-respuesta por tanto hubo que enfatizar en cual podría ser más eficiente en cada caso.
- Prueba de las comunicaciones: una vez implementada la solución, se tuvo que probar los diferentes casos de interacción, a excepción de las pruebas de los controladores REST para los que se utilizó Postman, la mayoría de estas pruebas fueron manuales dado que se había considerado que con tantos elementos en juego, el diseño y elaboración de estás sería muy costosa tanto en tiempo como complejidad.

La extensión del tiempo en los puntos anteriores ha tenido como consecuencia tener que recortar en la parte de análisis y monitorización de vehículos y puntos de carga a través de ThingsBoard, donde solamente se han llegado a cubrir los aspectos básicos sobre como se extraen los datos y la creación de paneles. La idea inicial era que se pudiera acceder a un panel específico a cada dispositivo, seleccionando cualquiera de ellos desde la tabla o desde un marcador del mapa.

Los problemas se han presentado mayoritariamente en la fase de implementación tanto por complejidad como por limitaciones temporales. Hubo diferentes problemas a nivel técnico, primero por la ejecución de Kubernetes(Minikube) en Windows, debido a conflictos entre los hipervisores Hyper-V y VirtualBox. La solución a esto fue realizar la instalación dentro de WSL, si bien ya no habían problemas con la ejecución no fue hasta que se intentó desplegar la solución que estos volvieron a surgir. La razón fue que los los pods eran incapaces de establecer comunicaciones a pesar de haber expuesto sus respectivos despliegues como servicios. Se ha puesto especial atención en que los nombres proporcionados para los servicios y los puertos fueran los mismos que han pasado como variables dentro de los despliegues. Es posible que se pueda evitar este tipo de problemas en una distribución de Linux como Ubuntu. Tampoco se ha tenido tiempo suficiente como para estudiar la posibilidad de probar el despliegue de la solución en un proveedor en la nube. Estas mismas limitaciones temporales también afectaron el despliegue utilizando Docker Swarm, con suficiente tiempo cumplir este objetivo habría sido posible.

En conclusión, se han alcanzado cinco de los séis objetivos que nos habíamos planteado, sin embargo, el trabajo no está excento de puntos débiles como las pruebas o la puesta en producción.

8.1 Relación del trabajo desarrollado con los estudios cursados

Los estudios cursados han sido una pieza vital para el desarrollo de la carrera, ya que se han utilizado los conocimientos y experiencia adquiridos de la programación en Java, aplicación de patrones y diseño de arquitectura software. A lo largo de la carrera también se han tratado temas como el diseño de sistemas distribuido y patrones de comunicación. En las distintas etapas de este trabajo se ha requerido de capacidad de comprensión, análisis, planificación y gestión del tiempo así como de pensamiento crítico que han aplicado de manera continua durante la realización del grado.

8.2 Trabajos futuros

Como primera propuesta de trabajo futuro tenemos el objetivo que no se ha podido alcanzar, el despliegue con Kubernetes. Aprovechando las capacidades de este orquestador se puede probar también el funcionamiento de ThingsBoard en su versión monolítica y la basada en microservicios bajo altas cargas de trabajo. Otra propuesta sería la incorporación de Kafka al sistema y comparar el rendimiento antes y después.

Para analizar el redimiento del sistema se podría introducir Prometheus y elaborar uno o varios paneles en Grafana para mostrar la información de una manera visual.

En entornos de producción reales también es importante mejorar la seguridad de las comunicaciones, ejemplo de esto sería agregar unas credenciales de acceso al bróker EMQX o hacer que los vehículos y puntos se autentiquen mediante JWT con el servicio de gestión.

Bibliografía

[1] Mauro A. A. da Cruz, Joel José P. C. Rodrigues, Jalal Al-Muhtadi, Valery V. Korotaev y Victor Hugo C. de Albuquerque, .ª Reference Model for Internet of Things Middleware", *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 871-883, 2018, DOI: 10.1109/JIOT.2018.2796561.

[2] Cockburn, Alistair. Hexagonal Architecture. Consultado el 9 de julio de 2023. Recuperado de: https://alistair.cockburn.us/hexagonal_architecture/

[3] Tom Hombergs *Get Your Hands Dirty on Clean Architecture* Packt Publishing, primera edición, 2019, 1839211962

[4] Kevin Ashton Invents the Term "The Internet of Things". Consultado el 20 de junio de 2023. Recuperado de: https://www.historyofinformation.com/detail.php?id=3411

[5] M2M. Consultado el 20 de junio de 2023. Recuperado de: https://en.wikipedia.org/wiki/Machine_to_machine

[6] Azure IoT. Consultado el 20 de junio de 2023. Recuperado de: https://azure.microsoft.com/es-es/solutions/iot/

[7] AWS IoT. Consultado el 20 de junio de 2023. Recuperado de: https://aws.amazon.com/es/iot/

[8] Watson IoT Platform. Consultado el 20 de junio de 2023. Recuperado de: https://www.ibm.com/es-es/cloud/watson-iot-platform/details

[9] ThingsBoard. https://thingsboard.io/

[10] OpenRemote. Consultado el 20 de junio de 2023. Recuperado de: https://www.openremote.io/

[11] Kaa Community. Consultado el 20 de junio de 2023. Recuperado de: https://kaaproject.github.io/kaa/docs/v0.10.0/Welcome/

[12] Electromaps. Consultado el 22 de junio de 2023. Recuperado de: https://www.electromaps.com/

[13] Smartmobility de Iberdrola. Consultado el 07 de septiembre de 2023. Recuperado de: https://www.iberdrola.com/es/empresas/soluciones/electromovilidad/smartmobility

[14] EVectrum. Consultado el 23 de junio de 2023. Recuperado de: https://evectrum.com/

[15] EVcharge. Consultado el 23 de junio de 2023. Recuperado de: https://evcharge.es/

[16] Chargepoint. Consultado el 23 de junio de 2023. Recuperado de: https://www.chargepoint.com/

[17] DispatchTrack. Consultado el 24 de junio de 2023. Recuperado de: https://dispatchtrack.com/

[18] APACHE LICENSE, VERSION 2.0. Consultado el 5 de julio de 2023. Recuperado de: https://www.apache.org/licenses/LICENSE-2.0

[19] GNU Affero General Public License. Consultado el 5 de julio de 2023. Recuperado de: https://www.gnu.org/licenses/agpl-3.0.html.en

[20] Eclipse Mosquitto. Consultado el 11 de julio de 2023. Recuperado de: https://mosquitto.org/

[21] VerneMQ. Consultado el 11 de julio de 2023. Recuperado de: https://vernemq.com/

[22] HiveMQ. Consultado el 11 de julio de 2023. Recuperado de: https://www.hivemq.com/

[23] EMQX. Consultado el 11 de julio de 2023. Recuperado de: https://www.emqx.io/

[24] Spring Boot. Consultado el 10 de julio de 2023. Recuperado de: https://spring.io/projects/spring-boot

[25] Spring Framework. Consultado el 10 de julio de 2023. Recuperado de: https://spring.io/projects/spring-framework

[26] Spring Initializr. Consultado el 10 de julio de 2023. Recuperado de: https://start.spring.io/

[27] Apache Maven. Consultado el 10 de julio de 2023. Recuperado de: https://maven.apache.org/

[28] Docker. Consultado el 12 de julio de 2023. Recuperado de: https://www.docker.com/

[29] Docker Overview. Consultado el 12 de julio de 2023. Recuperado de: https://docs.docker.com/get-started/

[30] Docker Compose. Consultado el 12 de julio de 2023. Recuperado de: https://docs.docker.com/compose/

[31] Docker Swarm. Consultado el 12 de julio de 2023. Recuperado de: https://docs.docker.com/engine/swarm/

[32] Kubernetes Community.
Kubernetes. Consultado el 13 de julio de 2023. Recuperado de: https://kubernetes.io/

[33] ThingsBoard Gateway. Consultado el 13 de julio de 2023. Recuperado de: https://thingsboard.io/docs/iot-gateway/

[34] MongoDB. Consultado el 14 de julio de 2023. Recuperado de: https://www.mongodb.com/

[35] Mongo Express. Consultado el 14 de julio de 2023. Recuperado de: https://github.com/mongo-express/mongo-express

[36] Visual Studio Code. Consultado el 14 de julio de 2023. Recuperado de: https://code.visualstudio.com/

[37] Postman. Consultado el 14 de julio de 2023. Recuperado de: https://www.postman.com/

[38] ThingsBoard Gateway. Consultado el 20 de agosto de 2023. Recuperado de: https://thingsboard.io/docs/user-guide/dashboards/

APÉNDICE A
OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenible	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**	X			
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**		X		
ODS 10. **Reducción de las desigualdades.**				X
ODS 11. **Ciudades y comunidades sostenibles.**		X		
ODS 12. **Producción y consumo responsables.**				X
ODS 13. **Acción por el clima.**	X			
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**		X		

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

El trabajo desarrollado tiene una clara relación con los Objetivos de Desarrollo Sostenible (ODS) de las Naciones Unidas, ya que aborda varios de los objetivos y contribuye de manera significativa a la promoción de la sostenibilidad en el sector del transporte y la logística. Los objetivos relacionados directamente con el trabajo son:

1. **ODS 7: Energía asequible y no contaminante**: El proyecto aborda directamente la promoción de una fuente de energía más sostenible al centrarse en la movilidad eléctrica. Al diseñar una solución que optimiza la carga de vehículos eléctricos, fomenta la adopción de esta tecnología, que se basa en electricidad en lugar de combustibles fósiles. Esto ayuda a reducir las emisiones de carbono y la contaminación del aire asociada con los vehículos de combustión interna, lo que es fundamental para lograr una energía más limpia y asequible.

2. **ODS 9: Industria, innovación e infraestructura**: El proyecto se basa en tecnologías computacionales modernas y principios de Internet de las Cosas para desarrollar una solución que mejora la infraestructura de carga y la eficiencia en la logística de vehículos eléctricos, lo que está en línea con el objetivo de promover la innovación tecnológica y el desarrollo de infraestructuras sostenibles.

3. **ODS 11: Ciudades y comunidades sostenibles**: El proyecto busca optimizar la logística de vehículos eléctricos en un escenario de reparto/logística en áreas urbanas. Al reducir la contaminación del aire y el ruido mediante la promoción de vehículos eléctricos y la mejora de la eficiencia en la entrega de bienes y servicios, contribuye directamente a la mejora de la calidad de vida en las ciudades.

4. **ODS 13: Acción por el clima**: el proyecto se alinea estrechamente con los esfuerzos para combatir el cambio climático. Al reducir las emisiones de gases de efecto invernadero y promover la adopción de vehículos eléctricos, contribuye a limitar el calentamiento global y sus efectos negativos. Esto es crucial en un momento en que la mitigación del cambio climático se ha vuelto una prioridad mundial. La mejora de la eficiencia en la logística de vehículos eléctricos también puede conducir a una reducción adicional de las emisiones al evitar desplazamientos innecesarios y optimizar las rutas de entrega.

5. **ODS 17: Alianzas para lograr los objetivos**: El proyecto involucra la colaboración entre diferentes sectores, incluidos el sector de la movilidad eléctrica, la tecnología de la información y la logística. La colaboración intersectorial es fundamental para abordar problemas complejos como la transición hacia una movilidad más sostenible. El proyecto demuestra cómo la colaboración entre estos sectores puede dar como resultado soluciones innovadoras y sostenibles.

En resumen, este proyecto se alinea estrechamente con varios de los Objetivos de Desarrollo Sostenible al promover la movilidad eléctrica, mejorar la infraestructura de carga, reducir las emisiones de gases de efecto invernadero y contribuir a la creación de ciudades más sostenibles. Además, la innovación tecnológica y la colaboración entre diferentes partes interesadas son elementos clave para lograr estos objetivos de manera efectiva. La implementación exitosa de este proyecto podría tener un impacto significativo en la transición hacia un transporte más sostenible y en la reducción de los impactos ambientales asociados con el sector del transporte y la logística.